université de BORDEAUX

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATHIQUES
ET INFORMATIQUE DE BORDEAUX
SPÉCIALITÉ INFORMATIQUE

Par David BONNIN

ALGORITHMIQUE DISTRIBUÉE
ASYNCHRONE AVEC UNE MAJORITÉ DE PANNES

Sous la direction de : Corentin TRAVERS et Yves MÉTIVIER

Soutenue le 24 novembre 2015

Membres du jury :

Président :		
PETIT Franck	Professeur des universités	UPMC
Rapporteurs :		
FERNÁNDEZ ANTA Antonio	Professeur des universités	IMDEA Networks Institute
MOSTÉFAOUI Achour	Professeur des universités	Université de Nantes
Directeurs de thèse :		
MÉTIVIER Yves	Professeur des universités	LaBRI
TRAVERS Corentin	Maître de conférences	LaBRI

Algorithmique distribuée asynchrone avec une majorité de pannes

Résumé :

En algorithmique distribuée, le modèle asynchrone par envoi de messages et à pannes est connu et utilisé dans de nombreux articles de par son réalisme, par ailleurs il est suffisamment simple pour être utilisé et suffisamment complexe pour représenter des problèmes réels. Dans ce modèle, les n processus communiquent en s'échangeant des messages, mais sans borne sur les délais de communication, c'est-à-dire qu'un message peut mettre un temps arbitrairement long à atteindre sa destination. De plus, jusqu'à f processus peuvent tomber en panne, et ainsi arrêter définitivement de fonctionner. Ces pannes indétectables à cause de l'asynchronisme du système limitent les possibilités de ce modèle.

Dans de nombreux cas, les résultats connus dans ces systèmes sont limités à une stricte minorité de pannes. C'est par exemple le cas de l'implémentation de registres atomiques et de la résolution du renommage. Cette *barrière* de la majorité de pannes, expliquée par le théorème CAP, s'applique à de nombreux problèmes, et fait que le modèle asynchrone par envoi de messages avec une majorité de pannes est peu étudié. Il est donc intéressant d'étudier ce qu'il est possible de faire dans ce cadre.

Cette thèse cherche donc à mieux comprendre ce modèle à majorité de pannes, au travers de deux principaux problèmes. Dans un premier temps, on étudie l'implémentation d'objets partagés similaires aux registres habituels, en définissant les *bancs de registres χ-colorés* et les *α-registres*. Dans un second temps, le problème du renommage est étendu en *renommage k-redondant*, dans ses versions *à-un-coup* et *réutilisable*, et de même pour les objets partagés diviseurs, étendus en *k-diviseurs*.

Mots clés : Algorithmes distribués, asynchrone, envoi de message, panne, mémoire partagée, registre, renommage, diviseur

Unité de Recherche

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
Université de Bordeaux
351, cours de la Libération
33405 Talence Cedex, France

Asynchronous distributed computing with a majority of crashes

Abstract :

In distributed computing, asynchronous message-passing model with crashes is well-known and considered in many articles, because of its realism and it is simple enough to be used and complex enough to represent many real problems. In this model, n processes communicate by exchanging messages, but without any bound on communication delays, i.e. a message may take an arbitrarily long time to reach its destination. Moreover, up to f among the n processes may crash, and thus definitely stop working. Those crashes are undetectable because of the system asynchronism, and restrict the potential results in this model.

In many cases, known results in those systems must verify the property of a strict minority of crashes. For example, this applies to implementation of atomic registers and solving of renaming. This *barrier* of a majority of crashes, explained by the CAP theorem, restricts numerous problems, and the asynchronous message-passing model with a majority of crashes is thus not well-studied and rather unknown. Hence, studying what can be done in this case of a majority of crashes is interesting.

This thesis tries to analyse this model, through two main problems. The first part studies the implementation of shared objects, similar to usual registers, by defining *χ-colored register banks*, and *α-registers*. The second part extends the renaming problem into *k-redundant renaming*, for both *one-shot* and *long-lived* versions, and similarly for the shared objects called *splitters* into *k-splitters*.

Keywords : Distributed algorithms, asynchronous, message-passing, crash, shared memory, register, renaming, splitter

Research Unit

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
Université de Bordeaux
351, cours de la Libération
33405 Talence Cedex, France

Remerciements

Je tiens tout d'abord à remercier Franck Petit, pour avoir accepté le poste de Président du jury de cette thèse. Merci aussi à Antonio Fernández Anta et Achour Mostéfaoui pour leur travail de rapporteur, dont la tâche de relire une thèse n'est pas de tout repos. Je remercie aussi Corentin Travers et Yves Métivier, mes directeurs de thèse, pour leur encadrement et leurs conseils tout au long de ces trois longues années de thèse.

Je souhaite aussi remercier les collègues que j'ai pu croiser au cours de cette thèse, qu'ils soient doctorants, docteurs ou ingénieurs, du LaBRI ou d'Inria, ainsi que le personnel technique et administratif, sans qui la vie au labo serait moins confortable. Je ne connais pas la moitié d'entre vous à moitié autant que je le voudrais ; et j'aime moins de la moitié d'entre vous à moitié aussi bien que vous le méritez, comme l'a écrit un certain Tolkien (et il s'agit bien d'un compliment). Merci en particulier à Romaric, Noël, Claire et Vincent, pour tous les bons moments passés ensemble, et merci aux autres que je ne me risquerai pas à tenter de tous citer, de peur d'en oublier et de ne m'en rendre compte que trop tard.

Merci à l'AFoDIB, au sein de laquelle j'ai effectué un travail de trésorier deux ans durant, pour la cohésion qu'elle apporte entre les doctorants informaticiens de Bordeaux, ainsi que les activités proposées, y compris celles permettant de décompresser après une journée de travail. Merci à l'ENSEIRB-MATMECA pour m'avoir accueilli en tant qu'enseignant, et aux étudiants pour leur indulgence envers mes capacités pédagogiques. Merci à son club JdR-JdS pour les week-end de détente autour de jeux divers et variés.

J'en viens maintenant à remercier ma famille et mes amis, pour leur soutien et encouragements, ainsi que la présence d'une partie d'entre eux à la soutenance de cette thèse, malgré leur probable incompréhension de l'exposé. *Last but not least*, un grand merci à Tamara, qui a su me soutenir et être présente pendant ces trente-sept mois de thèse, et notamment lors de la difficile rédaction du présent document. Enfin, je remercie à nouveau Tamara, Elio, Samuel, Jean-Pierre et Christine pour leur préparation du pot qui clôturera la soutenance de cette thèse.

Table des matières

1 Introduction **1**
1.1 L'algorithmique distribuée 1
1.1.1 Généralités 1
1.1.2 Le modèle asynchrone à pannes 2
1.1.3 Simulation de mémoire partagée 2
1.2 Limite à une majorité de pannes et partitionnement 3
1.2.1 Le théorème *CAP* 3
1.2.2 Barrière de la majorité de pannes 4
1.3 Problématique et contributions 5
1.3.1 De l'autre côté de la barrière 5
1.3.2 Extension des registres partagés 6
1.3.3 Étude du problème du renommage et des diviseurs 6
1.4 Organisation de ce document 6

2 Modèle et définitions **8**
2.1 Modèle 8
2.1.1 Processus 8
2.1.2 Communication par envoi de messages 9
2.1.3 Mémoire partagée 9
2.1.4 Synchronisme et asynchronisme 10
2.1.5 Défaillances 11
2.2 Problème, algorithme et exécution 11
2.2.1 Exécution 11
2.2.2 Algorithme et problème 12
2.3 Discussion sur le modèle asynchrone à pannes 13
2.3.1 Quelques propriétés 13
2.3.2 Pourquoi ces choix de modèle 13
2.4 Registres 14
2.4.1 Critère de cohérence et type de données 14
2.4.2 Nombre de lecteurs et d'écrivains 14
2.4.3 Registres à *test-and-set* et *compare-and-swap* 15
2.4.4 Puissance relative des registres 15

3 Analyse de l'existant **16**
3.1 Simulation de mémoire partagée 16
3.1.1 La simulation *ABD* et les quorums 16
3.1.2 Algorithme ABD et démonstrations 17
3.1.3 Barrière de la majorité de pannes et théorème CAP 20
3.2 Ce qui est fait pour contourner la barrière 21
3.2.1 Approche probabiliste 21
3.2.2 Cohérence finale 22
3.2.3 Cohérence causale et cohérence pipeline 23

3.3 Divers problèmes et résultats classiques . . . 24
3.3.1 Consensus et k-accord ensembliste . . . 24
3.3.2 Renommage et diviseurs . . . 25
3.3.3 Accord approximatif . . . 25

4 Extension des registres partagés . . . 27
4.1 Affaiblissement de la disponibilité : les registres χ-colorés . . . 27
4.1.1 Définition . . . 27
4.1.2 Graphes de Kneser et quorums colorés . . . 28
4.1.3 Implémentation . . . 30
4.1.4 Limites et utilisation . . . 33
4.2 Affaiblir la cohérence : les α-registres . . . 36
4.2.1 Motivation et inspirations . . . 36
4.2.2 Définition . . . 38
4.3 Simuler des α-registres . . . 39
4.3.1 Version bruyante . . . 39
4.3.2 Version silencieuse . . . 47
4.4 Borne inférieure sur les possibilités de simulation d'α-registres . . . 54
4.4.1 Première borne . . . 54
4.4.2 Amélioration de cette borne . . . 56
4.4.3 Et au-delà ? . . . 58
4.5 Composer des α-registres . . . 58
4.5.1 Un α-registre multi-écrivains ? . . . 58
4.5.2 Vecteur d'α-registres . . . 59
4.6 Remarques avant le chapitre suivant . . . 59

5 Extension d'autres objets partagés : les diviseurs et le renommage . . . 61
5.1 Les diviseurs habituels et leur application au renommage . . . 61
5.1.1 Renommage classique . . . 61
5.1.2 Diviseurs à-un-coup . . . 68
5.1.3 Renommage à base de diviseurs . . . 71
5.1.4 Diviseurs réutilisables . . . 73
5.1.5 Renommage réutilisable . . . 77
5.2 Extension du renommage et des diviseurs à-un-coup . . . 81
5.2.1 Lien avec les résultats du chapitre précédent . . . 81
5.2.2 Définitions . . . 82
5.2.3 Implémentation des k-diviseurs à-un-coup . . . 84
5.2.4 k-renommage à-un-coup . . . 87
5.3 Extension du renommage et des diviseurs réutilisables . . . 89
5.3.1 Définitions . . . 89
5.3.2 Implémentation des k-diviseurs réutilisables . . . 90
5.3.3 k-renommage réutilisable . . . 95

6 Conclusion . . . 101
6.1 Bilan . . . 101
6.2 Perspectives et ouvertures . . . 103

Bibliographie . . . 105

Glossaire . . . 111

Table des figures

2.1 Un système distribué de 4 processus muni d'un graphe complet de canaux de communication 9
2.2 Exemple d'exécution illustrant la propriété de *linéarisabilité* 10

4.1 Graphe de Kneser $KG_{5,2}$ 29
4.2 Coloration des quorums pour $n = 4$ et $f = n - f = 2$ 29
4.3 Utilisation des bancs de registres χ-colorés ($\chi = 3$) 36
4.4 Fonctionnement de *Accept* 40
4.5 Propriété de *Accept* 41
4.6 Visualisation des notations 44
4.7 Fonctionnement des *cles* 49
4.8 Exécution pour la démonstration de la borne inférieure 54

5.1 Propriétés d'un diviseur 69
5.2 Implémentation d'un diviseur à-un-coup 69
5.3 Grille de diviseurs pour le renommage, de largeur 5 72
5.4 Exécution possible sur un diviseur réutilisable 74
5.5 Implémentation d'un diviseur réutilisable 75
5.6 Exécution possible d'un renommage réutilisable 78
5.7 Grille de diviseurs pour le renommage réutilisable 79
5.8 Implémentation d'un k-diviseur à-un-coup 85
5.9 Implémentation d'un k-diviseur réutilisable 91

Liste des Algorithmes

3.1 Simulation ABD de registre atomique 17
4.1 Implémentation d'un banc de χ registres colorés 31
4.2 Exemple d'algorithme à registre atomique illustrant un problème possible du passage aux registres χ-colorés 34
4.3 Algorithme bruyant d'implémentation d'α-registre 39
4.4 Algorithme silencieux d'implémentation d'α-registre 47
5.1 Renommage à-un-coup, par envoi de messages 62
5.2 Implémentation d'un diviseur à-un-coup 69
5.3 Renommage à-un-coup à base de diviseurs 71
5.4 Implémentation d'un diviseur réutilisable 75
5.5 Renommage réutilisable à base de diviseurs 78
5.6 Implémentation de k-diviseur à-un-coup 84
5.7 Implémentation de k-diviseur réutilisable 91
5.8 Association de l'implémentation des k-diviseurs réutilisables d'une grille . . 96
5.9 Renommage k-redondant réutilisable à base d'une grille de k-diviseurs . . . 98

Chapitre 1

Introduction

1.1 L'algorithmique distribuée

1.1.1 Généralités

Avant de rentrer dans les détails du sujet de cette thèse, il est nécessaire de comprendre le domaine auquel elle appartient, appelé *algorithmique distribuée*. L'algorithmique distribuée est l'étude d'algorithmes, et de résolution de problèmes, dans un système dit *distribué*, c'est-à-dire composé de plusieurs entités indépendantes tentant de travailler ensemble pour atteindre un objectif commun. Ces entités sont généralement appelées *processus*, mais peuvent représenter n'importe quelle forme d'objet capable d'effectuer des opérations simples (calculs mathématiques, stockage et récupération locale de données, test et boucle), que ce soit un ordinateur au sein d'un réseau, un coeur de processeur, une machine de Turing, ou même un être humain. Ces entités, bien qu'indépendantes, peuvent communiquer entre elles par l'intermédiaire d'une interface de communication, afin de collaborer pour un but commun.

La particularité des systèmes distribués, et ce en quoi ils diffèrent de certains autres systèmes qui sont aussi répartis sur plusieurs entités, est qu'il n'y a pas de centralisation ni de processus dirigeant ou coordonnant les autres processus. Les processus du système n'ont qu'une connaissance *partielle* de l'état du système, et ne peuvent effectuer que des actions *locales*, en ayant pourtant un objectif *global*. Cette absence d'autorité centrale, présente dans de nombreux systèmes informatisés, est une caractéristique essentielle des systèmes distribués. Un algorithme séquentiel très long dont les calculs sont répartis parmi plusieurs machines afin de gagner en efficacité relève de la *parallélisation*, et n'est pas du domaine de l'algorithmique distribuée, à cause de cette centralisation et de la répartition entièrement contrôlée des tâches. Cette indépendance des processus et cette absence d'un contrôle *global* font que l'étude des problèmes et algorithmes distribués ne peut pas être une simple extension de celle des problèmes et algorithmes *séquentiels*.

À cause de ce manque de contrôle et d'informations *globales*, et de l'impossibilité de simplement étendre les connaissances actuelles sur les algorithmes séquentiels, on est en droit de se demander si ce modèle *distribué* est réellement utile. Un des exemples les plus probant d'application de l'algorithmique distribuée est probablement celui d'Internet. Internet est un immense réseau de machines diverses, chacune fonctionnant indépendamment, communiquant entre elles, et tâchant d'accomplir des buts communs, dont celui largement étudié du *routage* visant à transmettre efficacement des données [83], en indiquant à chaque intersection la direction à prendre pour atteindre une destination. Toujours lié à internet, certains sites populaires ne peuvent pas se contenter d'être basés sur un seul serveur, car une seule machine n'aurait pas la puissance et la rapidité nécessaires pour répondre aux demandes de clients en temps voulu, et disposent donc d'un réseau de ser-

veurs travaillant de concert afin de fournir des services rapides aux nombreux clients y accédant. Plus précisément, un client interagit avec un des serveurs, et ce serveur répond rapidement au client, tout en communiquant plus lentement avec les autres serveurs, afin de partager les données fournies par le client, notamment dans l'éventualité d'une future connexion entre ce client et un autre serveur du système. Enfin, un réseau distribué est moins sensible aux pannes qu'un réseau centralisé, car un processus dirigeant qui tombe en panne entraîne un blocage du réseau complet, alors qu'une panne dans un des processus égaux d'un système distribué peut être anticipée et avoir des conséquences plus minimes.

1.1.2 Le modèle asynchrone à pannes

L'algorithmique distribuée est un vaste domaine, et certains problèmes et surtout chaque algorithme que l'on peut y trouver est rattaché à un modèle particulier. En effet, il n'est pas possible de décrire un algorithme sans préalablement émettre quelques hypothèses sur la structure du système distribué, son interface de communication, et la présence potentielle de défaillances. Pour ces raisons, et afin de mieux comprendre la problématique de cette thèse, il est nécessaire de décrire le modèle qui sera utilisé ici.

Le principal modèle que l'on considère est un modèle par envoi de messages, asynchrone, et avec présence de pannes. Un système distribué fonctionnant par *envoi de messages* est un système dans lequel les processus peuvent communiquer entre eux en s'envoyant directement des messages d'un processus à l'autre. Un système est dit *asynchrone* si les durées de communications ne sont pas fixées ni même bornées : chaque message envoyé peut mettre un temps arbitrairement grand ou petit à arriver. Enfin, un système à *pannes* est un système dans lequel certains processus peuvent, à tout moment, tomber en panne sans prévenir et ne plus fonctionner ni exécuter d'algorithme.

Ce modèle est très utilisé dans le domaine [21, 44, 17, 16, 39, 80, 86, 42], car représentant assez fidèlement des systèmes réels (réseaux informatiques), tout en restant pratique à manipuler. En effet, les défaillances par *pannes* sont courantes dans tout système informatisé, et plus probables dans de larges systèmes distribués à cause du nombre élevé d'entités présentes. Les systèmes distribués usuels fonctionnent généralement à l'aide d'envois de messages, sous une forme ou une autre, par exemple par paquets TCP/IP. De plus l'asynchronisme est une condition suffisamment générale pour couvrir la majorité des spécificités de communications dans les cas pratiques. En effet, même les systèmes synchrones peuvent avoir un comportement similaire en présence d'erreurs dans les canaux de communication, car en absence de confirmation de réception d'un message, il est courant de l'envoyer à nouveau jusqu'à qu'il arrive correctement à destination.

Enfin, ce modèle a quelques limites qu'il semble important de souligner ici. En effet, la présence de pannes empêche l'utilisation d'élection d'un *chef* afin de centraliser l'algorithme plus simplement, car si ce chef tombe en panne, le système risque de ne plus fonctionner. Un autre problème est la combinaison de l'asynchronisme avec la présence de pannes. En effet, du point de vue d'un processus A, si le processus B ne répond pas à ses messages au bout d'un certain temps, il n'est pas possible de savoir si B est tombé en panne, ou bien si l'acheminement des messages entre les 2 processus est arbitrairement lent.

1.1.3 Simulation de mémoire partagée

Bien que ce système asynchrone par envoi de messages et avec pannes ait ses limites, de nombreux problèmes peuvent y être résolus. L'un des résultats les plus importants de ce modèle est celui de la simulation de registres partagés par Attiya, Bar-Noy et Dolev, en 1995 [16]. En effet, le modèle par *mémoire partagée* est un autre modèle d'inter-

face de communication pour des systèmes distribués, qui est assez fréquemment utilisé [21, 59, 37, 3, 78, 58, 42], et principalement par l'intermédiaire de *registres* partagés. Ce modèle, représentant mieux les architectures internes des processeurs multicoeurs et autres ordinateurs multiprocesseurs, a aussi l'avantage d'être communément plus simple à utiliser dans la conception d'algorithmes distribués, car l'utilisation de registres de mémoire partagés est plus intuitive (pour certains) que celle de messages, et conduit généralement à des algorithmes plus facilement compréhensibles.

La simulation de registres partagés dans notre modèle implique que tout algorithme écrit pour un modèle basé sur des registres partagés peut immédiatement être transcrit en algorithme fonctionnant dans le modèle asynchrone à envoi de messages et avec pannes. Ce lien entre les deux modèles fait que l'on peut toujours se considérer dans le modèle par envoi de messages, plus général, pour ce qui est de savoir ce qu'il est possible de faire, et c'est ce que l'on fera donc par la suite dans cette thèse. Ainsi, tout algorithme réalisé dans un modèle asynchrone utilisant des registres partagés plutôt que des envois de messages est aussi, directement, un algorithme dans notre modèle.

Cependant, il est à noter que cette simulation présente une condition importante qui est l'existence d'une borne limitant le nombre possible de pannes à une minorité stricte des processus. En effet, dès que le nombre possible de pannes parmi les processus est supérieur ou égal à la moitié des processus, alors cette simulation n'est plus possible.

1.2 Limite à une majorité de pannes et partitionnement

1.2.1 Le théorème *CAP*

Le théorème *CAP*, présenté comme conjecture lors de la conférence PODC 2000 par Eric Brewer, puis prouvé en 2002 par Gilbert et Lynch [54], est un résultat important pour la conception d'algorithmes robustes dans des systèmes distribués. Ce théorème affirme qu'il est impossible pour un système distribué de garantir en même temps les 3 propriétés suivantes :

- *Cohérence* des données (*Consistency* en anglais) : tout processus tentant d'accéder à des données partagées du système doit obtenir des données cohérentes par rapport à ce que peuvent obtenir les autres processus dans le même temps. Bien que la notion de *données cohérentes* puisse différer d'un modèle à l'autre, la définition la plus courante est cette d'atomicité (ou linéarisabilité) des opérations, définie dans le chapitre suivant. Informellement, cette propriété demande à ce que chaque lecture d'un objet de mémoire partagée retourne la dernière valeur écrite dans cet objet.
- *Disponibilité* (*Availability* en anglais) : toutes les requêtes, ou opérations, invoquées par des processus doivent terminer et aboutir à une réponse.
- *Tolérance au partitionnement* (*Partition tolerance* en anglais) : dans le cas où le système se retrouve séparé en plusieurs parties indépendantes, les processus de chaque partie doivent pouvoir continuer à fonctionner correctement. Cette propriété est liée aux deux précédentes pour ce qui est de définir *fonctionner correctement*, et un système qui garantirait les 3 propriétés devrait continuer à vérifier les propriétés de cohérence et de disponibilité même en présence de partitionnement.

Plus précisément, ce théorème se réfère à des serveurs web permettant à des clients d'accéder à certains services (bien que pouvant potentiellement s'appliquer à d'autres systèmes). Un tel serveur virtuel est composé de plusieurs machines, de sorte qu'un client effectuant une requête s'adresse à une de ces machines, afin de ne pas surcharger un seul

serveur. Un tel réseau de serveurs stocke des données, et chaque client (par l'intermédiaire d'une machine) peut accéder ou modifier en temps réel certaines de ces données, justifiant ainsi la demande de cohérence et de disponibilité des données.

Le problème des partitionnements est un problème courant dans certains de ces systèmes requérant une certaine rapidité de réponse, tels les larges bases de données comme celle du site internet Amazon (Dynamo [41]) ou Facebook (Cassandra [65]) par exemple. En effet, les machines du réseau de serveur ne peuvent pas toujours se permettre (pour des questions de délais) de communiquer avec suffisamment d'autres machines-serveurs pour garantir une absence de partitionnement. D'autre part, ce phénomène peut aussi se produire dans un système pouvant subir de nombreuses défaillances, et aussi sur des systèmes dont les différentes entités sont réparties entre plusieurs groupes géographiquement éloignés et ayant ainsi des communications plus lentes et potentiellement défaillantes entre eux. Ainsi, si plusieurs groupes de processus se retrouvent séparés les uns des autres à cause d'une trop grande latence dans les communications, ou de défaillances plus importantes, il est préférable que chaque *partition* soit capable de continuer son oeuvre indépendamment des autres (par exemple [89] présente des résultats dédiés aux systèmes géo-distribués).

Cependant, bien que la tolérance au partitionnement soit utile, il peut aussi être coûteux de perdre en cohérence ou en disponibilité pour trouver un compromis. En effet, sans disponibilité, certains processus peuvent attendre une réponse pour toujours, et donc ne plus exécuter d'autre tâche, et dans des cas pratiques, les clients utilisant un service sur un site Internet n'obtiendraient aucune réponse et verraient leur page bloquée. D'autre part, un manque de cohérence peut lui aussi avoir des conséquences néfastes, si un processus a des informations obsolètes au sujet du système global, il risque d'avoir un comportement incorrect. Par exemple, si, pour une base de données représentant une gestion de stock de certaines marchandises, un processus a des données obsolètes correspondant à une quantité disponible plus élevée que la normale pour un certain objet, un client pourrait demander à obtenir plus de ces objets qu'il n'y en a effectivement en stock.

Enfin, il est important de noter que, bien que ce théorème énonce l'impossibilité d'un système parfait qui respecterait ces 3 conditions, il est possible de nuancer les pertes d'une (ou plusieurs) de ces conditions. En effet, un système garantissant une tolérance au partitionnement et une disponibilité ne peut pas garantir une cohérence totale, mais cela ne signifie pas qu'il n'est pas possible de garantir certains critères de cohérence *faible*. Ainsi, certains systèmes utilisent la notion de *cohérence finale* (*eventual consistency* en anglais) qui garantit une stabilisation des données, dans le sens où, au bout d'un certain temps sans ajout de données (nouvelles valeurs écrites), chaque processus du système partage les même données ([94, 28], voir partie 3.2.2). Certains travaux cherchent par ailleurs à aider un programmeur à mettre en place des systèmes vérifiant différents compromis entre ces 3 propriétés, ces compromis pouvant varier d'un aspect à l'autre d'un large système [95, 93].

1.2.2 Barrière de la majorité de pannes

Le théorème CAP trouve bien sa place dans le modèle défini précédemment. En présence d'un majorité de pannes potentielles, c'est-à-dire si au moins la moitié des processus *pourraient* tomber en panne lors de l'exécution d'un algorithme, un partitionnement est possible. En effet, si les processus sont séparés en 2 groupes égaux, et que les messages d'un groupe à l'autre sont ralentis arbitrairement longtemps (dû à l'asynchronisme), alors il est impossible pour les processus d'un groupe de savoir si ceux de l'autre groupe sont tombés en panne ou s'ils sont effectivement ralentis. De cette manière, si un algorithme se veut résistant au partitionnement et vérifie la propriété de disponibilité, chacun des groupes doit continuer à exécuter son algorithme indépendamment de l'autre, comme si

celui-ci était tombé en panne.

D'une certaine manière, ce théorème explique la condition évoquée précédemment pour la simulation de mémoire partagée sur la limite à une minorité de pannes. En effet, les registres partagés qui sont simulés dans cet algorithme vérifient de fortes propriétés de cohérence et de disponibilité, et il n'est donc pas possible de les garantir tout en garantissant une tolérance au partitionnement. Plus précisément, ces registres (définis dans la partie 2.1.3) vérifient que leurs opérations terminent, et que chaque lecture retourne la dernière valeur écrite, en accord avec la *linéarisabilité*. C'est pourquoi cette limite est importante, car elle assure que le système ne subira pas de partitionnement, si l'algorithme est correctement conçu pour tenir compte de ces pannes (un algorithme ne tenant pas compte des pannes pourrait restreindre ses communications, fonctionnant ainsi sans panne mais subissant une forme de partitionnement dans le cas contraire).

Ce lien entre partitionnement et majorité de pannes a aussi des conséquences plus générales sur ce modèle. Si l'on souhaite écrire un algorithme pour un tel système ayant potentiellement plus d'une majorité de pannes, alors le partitionnement est possible et il faut donc prévoir une tolérance à ce dernier. Ce qui implique que tout algorithme dans un tel système ne peut pas garder à la fois des propriétés de cohérence et de disponibilité inchangées. Enfin, cela explique pourquoi ce modèle est souvent considéré avec une stricte minorité de pannes.

1.3 Problématique et contributions

1.3.1 De l'autre côté de la barrière

Cette *barrière* d'une majorité de pannes sépare deux domaines du monde du distribué proches mais différents. Tandis que du côté de la minorité de pannes, ce modèle est bien connu et étudié, notamment par l'intermédiaire du modèle à mémoire partagée, le modèle avec majorité de pannes est méconnu et les résultats y sont bien moins nombreux. Dans le premier cas, la conception d'algorithmes et l'analyse de problèmes sont plus cadrées et systématisées, par exemple par l'utilisation d'abstractions telles que le *snapshot* [3, 32], *adopt-commit* [49], ou encore par les méthodes de preuve utilisant des propriétés de topologie [20, 77]. Par ailleurs, des propriétés caractérisent la calculabilité (c'est-à-dire ce qu'il est possible de faire) dans ce modèle, qui est équivalente à la calculabilité dans le modèle à mémoire partagée (au lieu de l'envoi de messages). Alors que dans le second cas, l'écriture d'algorithmes demande plus de réflexion et de choix, notamment pour trouver des compromis entre disponibilité et cohérence. En effet, comme il devient nécessaire de conserver la tolérance au partitionnement, il convient d'affaiblir la disponibilité ou la cohérence du problème, mais supprimer (ou tout simplement *trop* affaiblir) une de ces propriétés amène des résultats inintéressants et peu utilisables.

Enfin, il existe des cas pratiques pour lesquels le partitionnement est un réel problème, notamment les systèmes *géo-distribués*, c'est-à-dire les réseaux dont les éléments sont répartis en plusieurs groupes, chacun séparé d'un autre par de grandes distances à l'échelle mondiale. Ces grandes distances font que les communications d'un groupe vers un autre peuvent être compliquées, car ralenties ou interrompues avec plus de risques que les communications internes à un groupe. Pour toutes ces raisons, il est intéressant de se pencher sur ce modèle avec une majorité de pannes, afin d'essayer d'en analyser les difficultés et de découvrir ce qu'il est possible d'y faire. L'objectif de cette thèse est donc de développer des objets partagés pouvant potentiellement faciliter l'écriture d'algorithme dans ce modèle, à la manière des registres, et d'étudier le comportement de certains problèmes classiques dans ce modèle.

Après ce chapitre d'introduction, les chapitre 2 présentent le modèle utilisé et définit

formellement ses principaux concepts. Puis le chapitre 3 détaille des résultats existants et des approches liées à cette problématique. Enfin, les chapitres 4 et 5 entrent plus en détails dans les résultats de cette thèse.

1.3.2 Extension des registres partagés

Le chapitre 4 présente une extension des registres partagés dans ce modèle à majorité de pannes. La simulation de registres classiques dans le modèle avec une minorité de pannes [16] fait que tout résultat obtenu à l'aide de ces registres se traduit automatiquement en un résultat dans le modèle asynchrone par envoi de messages avec une minorité de pannes. Et comme les registres habituels sont couramment utilisés, mais ne peuvent pas être implémentés dans un système avec une majorité de pannes, il semble naturel de chercher à relâcher les propriétés de ces registres pour créer des extensions plus faibles de registres qui seraient simulables malgré une majorité de pannes.

Dans cette optique, nous avons eu deux approches principales pour cette extension des registres. La première idée est de relaxer la propriété de terminaison, et ainsi affaiblir la *disponibilité* du registre, tout en laissant un certaine utilisation possible en combinant plusieurs de ces registres : c'est ce que nous avons appelé les *registres colorés*. La seconde idée est d'affaiblir les critères de cohérence en permettant de lire plusieurs valeurs parmi celles précédemment écrites, dans la limite de α valeurs obsolètes : il s'agit des α-registres.

1.3.3 Étude du problème du renommage et des diviseurs

Le chapitre 5 étudie à la fois un objet partagé qui peut servir d'abstraction pour certains algorithmes (principalement de renommage), appelé *diviseur*, et un problème classique qu'est le *renommage*. Le renommage est un problème connu et largement étudié ([37] présente un bon état de l'art) qui, outre ses propriétés intéressantes d'un point de vue théorique, a des applications pratiques, comme par exemple dans le domaine de l'allocation de ressources.

Les diviseurs sont des objets introduits pour résoudre le renommage [78], et pouvant être implémentés à partir de registres partagés, donc dans le modèle asynchrone par envoi de messages avec une minorité de pannes. Le renommage [17] peut lui aussi être résolu dans ce modèle.

Nous avons donc étendu le problème du renommage à une notion plus générale de renommage k-redondant, qui peut être résolue dans notre modèle à majorité de pannes. Nous avons aussi étendu cette notion de diviseur à celle de *k-diviseur*, une extension présentant des propriétés semblables mais pouvant être simulée dans un modèle avec une majorité de pannes, et servant notamment à résoudre le renommage k-redondant.

1.4 Organisation de ce document

Cette thèse a donné lieu aux deux principales publications suivantes : [29] et [30]. Elle est présentée de la manière suivante :

- Le chapitre 1 est le chapitre actuel, qui introduit le sujet de cette thèse et sa problématique.
- Le chapitre 2 présente plus formellement le modèle qui sera considéré, et définit les différentes notions et notations utilisées. Le modèle y est notamment discuté, et quelques variations de ce dernier sont présentées à titre informatif.
- Le chapitre 3 présente des résultats connus dans ce domaine de l'algorithmique distribuée. On y trouve notamment des problèmes classiques du distribué, dont la simulation ABD, détaillée pour expliciter la barrière de la majorité de pannes.

D'autres travaux connexes à la notion de dépasser la barrière de la majorité sont aussi évoqués dans ce chapitre.

— Le chapitre 4 traite de l'extension des registres de mémoire partagée, dans le cadre de leur implémentation en présence d'une majorité de pannes. La première approche, s'inspirant de la simulation ABD, affaiblit la propriété de disponibilité pour définir des *bancs de registres χ-colorés*, pouvant être implémentés avec $\chi \geq 2f-n+2$. La seconde approche réduit les propriétés de cohérence en définissant des *α-registres*, implémentés pour $\alpha \geq 2(2f - n + 2) - 1$, et on prouve qu'ils ne peuvent pas être implémentés avec $\alpha \leq 2f - n + 2$.

— Le chapitre 5 discute des extensions des *diviseurs* et du *renommage*. On y définit les k-diviseurs à-un-coup et réutilisables, ainsi que le renommage k-redondant à-un-coup et réutilisable. Ces deux objets sont implémentés avec $k = \lfloor n/(n-f) \rfloor$, qui est démontré optimal. Les deux problèmes correspondants sont eux aussi résolus avec $k = \lfloor n/(n-f) \rfloor$, qui est aussi optimal.

— Le chapitre 6 conclue cette thèse en présentant un bilan des différents résultats, ainsi que des ouvertures possibles pour étendre ces travaux.

La bibliographie présente ensuite les références des divers travaux cités dans ce document, ordonnées par ordre alphabétique d'auteurs.

Enfin, un glossaire permet de rappeler succinctement les définitions des diverses notations et mots de vocabulaire spécifiques utilisés au cours de cette thèse, ainsi que des indications de pages pertinentes où ils sont utilisés et définis.

Chapitre 2

Modèle et définitions

Ce chapitre présente le modèle utilisé dans cette thèse, ainsi que des définitions des notions de problème, algorithme, et exécution, dans le cadre du distribué. Une discussion sur les choix du modèle est aussi présente, ainsi qu'une partie sur les différents types de registres partagés.

2.1 Modèle

Lorsqu'il est question d'étude d'un système distribué, on le considère toujours dans un modèle donné. Un tel modèle définit les différents aspects du système, c'est-à-dire les processus, l'interface de communication, ainsi que les aspects *temporels* et les éventuelles *défaillances*. Les notions définies dans cette partie sont des notions classiques qui peuvent être retrouvées dans la littérature sur le sujet, comme par exemple [21].

2.1.1 Processus

On considère un système distribué composé de $n \in \mathbb{N}^*$ entités appelées *processus*. L'ensemble de ces processus est noté $\mathcal{P}$, et on attribue à chaque processus, de manière arbitraire, un *indice* unique i, avec $1 \leq i \leq n$, de manière à noter p_i le $i$ème processus. On a alors $\mathcal{P} = \{p_1, \ldots, p_n\}$.

Les processus sont des machines de Turing, et sont donc capables de stocker des données dans une mémoire locale, et d'exécuter n'importe quel algorithme séquentiel *classique* (d'après la thèse de Church-Turing).

Chaque processus du système connaît le nombre n de processus, mais aucun ne connaît les *indices* i (et donc la notation p_i associée). En revanche, chaque processus dispose d'un identifiant unique noté $id(p_i)$ ou encore id_i, qui est un entier naturel quelconque. En particulier, si $p_i \neq p_j$, alors $id_i \neq id_j$. Initialement, chaque processus p_i connaît son propre identifiant $id(p_i)$, mais ne connaît pas ceux des autres processus.

Ce choix de ne donner qu'un identifiant arbitraire à chaque processus permet d'avoir un modèle plus général et plus réaliste (plutôt que de supposer que les processus sont identifiés par leurs indices dans $[1..n]$), et la notation p_i n'existe que pour le confort et la clarté de l'écriture d'algorithmes distribués et de preuves de ces algorithmes. Certains modèles, que nous ne considérerons pas dans ce document, généralisent d'avantage en empêchant les processus d'avoir des identifiants (système dit *anonyme*), ou en laissant le nombre n de processus inconnu de ces derniers [14, 38].

2.1.2 Communication par envoi de messages

Le modèle de communication utilisé dans ce document, sauf mention contraire, sera celui par envoi de messages présenté ici.

Un tel modèle est défini par un graphe dont les sommets sont les processus du système et les arêtes représentent des canaux de communication entre les processus. On ne s'intéressera dans cette thèse qu'aux modèles par envoi de messages dont le graphe est *complet* avec des canaux unidirectionnels, c'est-à-dire que pour chaque couple (p_i, p_j) de processus (y compris si $p_i = p_j$), il existe un canal de communication permettant à p_i d'envoyer des messages à p_j. La figure 2.1 permet d'illustrer un tel modèle, où les processus sont représentés par des cercles, et les canaux de communication par des flèches.

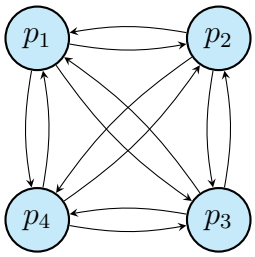

FIGURE 2.1 – Un système distribué de 4 processus muni d'un graphe complet de canaux de communication

On suppose que les canaux de communication sont *sûrs*, c'est-à-dire qu'il ne peut pas y survenir de modification ou disparition des messages envoyés, ni de création spontannée de message (y compris la duplication de messages envoyés). En particulier, tout message envoyé par p_i à p_j finira par être reçu, intact, par p_j. Enfin, on considère que les canaux sont *FIFO*, c'est-à-dire que si p_i envoie plusieurs messages à un processus p_j, ces messages seront reçus par p_j dans le même ordre qu'ils ont été envoyés. Ces choix de modèle seront justifiés dans la partie 2.3.

À titre informatif, il existe aussi d'autres modèles similaires de communication. Par exemple, le modèle par *agents mobiles* [70, 38] considère que les entités de calcul se déplace de nœud en nœud dans un graphe. Il s'agit en quelque sorte d'une variante du modèle par envoi de message, pour lequel les processus n'exécutent des parties non-triviales de leur programme qu'après réception d'un message et jusqu'à l'envoi d'un message à un autre processus, représentant ainsi le déplacement d'un agent.

2.1.3 Mémoire partagée

Dans un modèle par mémoire partagée, on considère que les processus ne peuvent pas s'envoyer de messages, mais communiquent plutôt par l'intermédiaire d'*objets* qui sont *partagés* par tous les processus du système. Ainsi, les processus peuvent accéder à un objet en utilisant des *opérations* sur cet objet.

Plus précisément, un objet est à tout instant dans un certain *état*, et chaque opération appelée par un processus sur cet objet est susceptible de modifier cet état, ainsi que de retourner au processus une valeur qui est une fonction de l'état de l'objet.

L'objet partagé le plus couramment utilisé est le *registre atomique*, associé à deux opérations : *lecture* et *écriture* (*read* et *write* en anglais). L'état du registre correspond, à tout instant, à la *valeur* qui est stockée dans ce registre, la valeur initiale étant notée $\perp$.

L'opération d'écriture prend une valeur en paramètre, et modifie l'état du registre en le remplaçant par cette valeur. L'opération de lecture ne modifie pas l'état du registre, mais retourne au processus qui l'a utilisé la valeur contenue dans le registre.

Enfin, comme chaque opération prend un temps variable entre son appel et sa terminaison, et comme plusieurs processus peuvent appeler concurremment des opérations sur le même objet, il est nécessaire de définir le comportement de ce registre dans le cas d'appels concurrents. Ainsi, un registre *atomique* vérifie la propriété de *linéarisabilité* [60], c'est-à-dire que, pour toute exécution possible, pour chaque instance d'opération op, il existe un instant t_{op} entre son appel et sa terminaison, tel que chaque lecture op_1 retourne la valeur écrite par l'écriture op_2 vérifiant $t_{op_2} < t_{op_1}$ avec le plus grand t_{op_2}, ou la lecture op_1 retourne $\perp$ si aucune écriture op_2 ne vérifie $t_{op_2} < t_{op_1}$.

En quelque sorte, la linéarisabilité représente le fait que chaque opération est globalement *vue* comme étant *atomique*, c'est-à-dire ne prenant qu'un instant pour s'exécuter, et l'ensemble des ces opérations (qui ne peuvent alors plus être concurrentes) vérifie les propriétés de cohérence habituelles, c'est-à-dire que chaque lecture retourne la dernière valeur écrite.

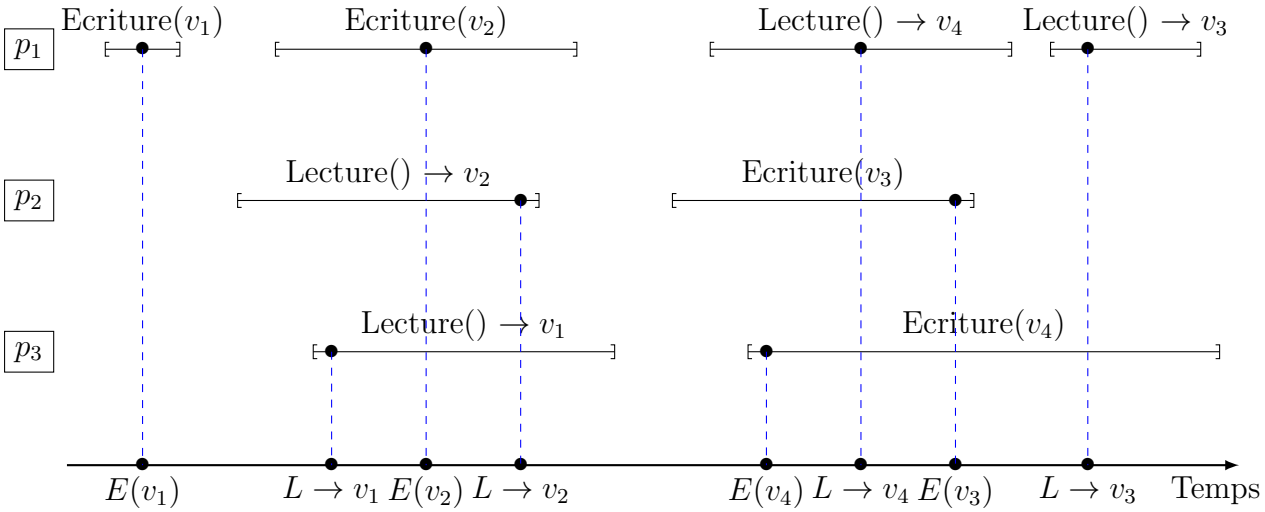

FIGURE 2.2 – Exemple d'exécution illustrant la propriété de *linéarisabilité*

La propriété de linéarisabilité est illustrée dans la figure 2.2. Les instants t_{op} sont représentés par les points noirs, qui sont totalement ordonnés dans le temps, et représentent l'exécution logique des différentes opérations. On peut notamment y remarquer que l'ordre logique des opérations concurrentes n'est pas forcément celui déterminé par l'ordre des appels et retours de fonctions.

Il est aussi à noter que certains objets partagés peuvent avoir des restrictions sur quels processus peuvent appeler quelles opérations, voire sur le nombre d'appels possibles aux opérations. Par exemple, il existe différentes catégories de registres atomiques, dont celle de *mono-écrivain* et *multi-lecteurs* (abrégé en 1WnR), pour laquelle un unique processus peut utiliser l'opération d'écriture sur le registre.

Enfin, d'autres objets partagés sont parfois considérés dans le modèle à mémoire partagé, comme par exemple des registres disposant de l'opération *test-and-set* ou encore *compare-and-swap*, plus puissants (en termes de calculabilité) que les registres à lecture-écriture habituels (voir partie 2.4.3).

2.1.4 Synchronisme et asynchronisme

En plus des interfaces de communication, un modèle de système distribué se doit de définir certains aspects que l'on qualifie de *temporels*. Ces aspects se restreignent principalement au choix entre *synchronisme* et *asynchronisme*, bien qu'il existe aussi des variantes comme le *synchronisme partiel*.

Dans le modèle *synchrone*, les communications se font en temps borné, et la borne T est connue des processus. Ainsi, les messages envoyés dans le modèle par envoi de messages sont reçus au plus tard un temps T après leur envoi. Dans le modèle à mémoire partagée, les opérations permettant d'accéder à des objets partagés mettent au plus un temps T à retourner après avoir été appelées. De plus, les temps d'exécution de calculs locaux par les processus sont eux aussi bornés par un temps T' connu des processus. Enfin, la présence et connaissance d'une telle borne permet de découper l'exécution d'un algorithme en séquence de *rondes* de durée $T + T'$, durant lesquelles les communications sont initiées en début de chaque ronde.

On considérera dans ce document des systèmes distribués dans le modèle *asynchrone*. Dans ce modèle, il n'existe pas de borne connue sur le temps de communication de l'interface, même si chaque communication (qu'il s'agisse d'un message envoyé ou d'un appel à une opération) prend un temps fini. En plus des temps de communication qui peuvent être arbitrairement grand, les vitesses d'exécution relatives des processus peuvent être arbitrairement variées, c'est-à-dire qu'un processus peut n'exécuter qu'un seul pas de calcul dans son algorithme pendant qu'un autre en exécute un million.

2.1.5 Défaillances

Les systèmes distribués peuvent être sujets à des défaillances sous une forme ou une autre. Un exemple de défaillance est celui des canaux de communication défaillants, et peut résulter en la perte ou modification de messages envoyés. On considérera ici que les systèmes de communication ne sont pas sensibles à de tels problèmes, et que seuls les processus peuvent être victimes d'erreurs, comme justifié dans la partie 2.3.

Un des modèles de défaillances de processus, et le seul utilisé dans ce document, est celui des *pannes*. Dans un tel modèle, un processus peut, au cours d'une exécution, spontanément tomber en panne, c'est-à-dire arrêter d'exécuter son algorithme, et ce de manière définitive. En particulier, une opération appelée par ce processus peut alors ne jamais terminer, et les messages qui devraient être reçus par ce processus sont perdus. Enfin, on suppose que le nombre de processus du système susceptibles de tomber en panne au cours d'une même exécution est borné par un nombre f, et que cette valeur est connue de tous les processus. Un processus qui ne tombe pas en panne au cours d'une exécution est dit *correct*.

À titre informatif, il existe notamment (parmi de nombreux autres) un autre modèle de défaillance similaire et connu, appelé le modèle *byzantin* [1, 90]. Un processus *byzantin* a un comportement arbitraire, et peut suivre son algorithme habituel comme suivre n'importe quel autre algorithme non souhaité. Ce modèle est plus général que celui à pannes, car un processus byzantin peut simplement décider de suivre l'algorithme pendant un temps avant d'arrêter subitement d'exécuter des pas de calcul, simulant ainsi une panne.

2.2 Problème, algorithme et exécution

2.2.1 Exécution

On appelle *événement* d'un processus p l'une des deux possibilités suivantes : le processus p exécute un pas de calcul local atomique (c'est-à-dire une *transition* de machine de Turing), ou le processus p exécute une *étape de communication*, dont la définition (ci-dessous) dépend du modèle utilisé. Un *événement* du système est alors un événement d'un des processus du système. Le système dispose d'une *horloge* globale qui permet d'ordonner de manière unique chacun des événements de ce système, mais les processus n'ont

pas accès à cette horloge. Dans un système distribué donné, une *exécution* est une suite (potentiellement infinie) d'*événements* ordonnés dans le temps par cette horloge.

Dans le modèle par envoi de messages, une *étape de communication* est soit l'envoi par un processus p_i d'un message m à un processus p_j, soit la réception par p_j d'un message m précédemment envoyé par un processus p_i à p_j. Dans le modèle par mémoire partagée, une *étape de communication* est soit l'*appel* à une opération *op* par un processus p sur un objet O, soit le retour d'une opération *op* sur un objet O par un processus p.

On considère que, pour toute exécution d'un système à mémoire partagée, un processus p qui a appelé une opération *op* sur un objet O ne peut plus appeler d'opération sur O jusqu'à ce que l'opération *op* retourne. De plus, seule une opération *op* ayant été appelée par un processus p peut retourner, et seulement auprès de ce même processus. On dit qu'une opération *termine* lorsqu'elle retourne auprès du processus qui l'a appelée. Enfin, on dira qu'au cours d'une exécution, une opération op_1 *précède* une opération op_2, si op_1 termine avant que op_2 ne soit appelé. Deux opérations op_1 et op_2 telles qu'aucune ne précède l'autre sont dites *concurrentes*.

2.2.2 Algorithme et problème

Un *algorithme distribué* est un ensemble de n *algorithmes locaux* (un par processus), chacun de ces algorithmes locaux étant un algorithme séquentiel classique mais disposant en outre de primitives de communication. Plus précisément, dans le modèle par envoi de messages, les primitives de communication permettent d'envoyer un message à un processus, et d'attendre la réception d'un message (éventuellement d'un certain type) ainsi que de le lire. Dans les modèle par mémoire partagée, les primitives de communication consistent à appeler une opération sur un objet, et à attendre le retour de cette opération pour en récupérer la valeur (ou simplement la confirmation de la fin de l'opération).

Il est à noter qu'il est possible pour un algorithme local d'exécuter plusieurs algorithmes en parallèle (à la manière du multithreading), par exemple de manière à avoir une gestion des messages reçus, sans pour autant bloquer le reste de l'algorithme. En particulier, cela permet aussi à un processus d'appeler plusieurs opérations sur des objets partagés différents en parallèle, mais il ne pourra pas ainsi effectuer plusieurs opérations simultanées sur un même objet, sauf si la spécification de l'objet en question l'autorise explicitement.

Un *problème* distribué est défini par un ensemble de *contraintes* $\mathcal{C}_e$ sur les valeurs d'entrée possibles, ainsi qu'un ensemble de *propriétés* $\mathcal{P}_r$ que doit vérifier le système durant la résolution de ce problème. Plus précisément, l'état global du système à un instant t d'une exécution est l'état à cet instant de chaque processus et de chaque canal de communication (messages envoyés mais pas encore reçus), et les propriétés de $\mathcal{P}_r$ sont des propriétés sur cet état global (et éventuellement sur les états globaux des instants passés), qui doivent être vérifiées à tout instant de l'exécution. Un *algorithme* distribué résout un problème distribué dans un modèle distribué donné si, pour tout ensemble de valeurs d'entrée I vérifiant les contraintes de $\mathcal{C}_e$, pour toute exécution de l'algorithme avec ces valeurs d'entrées dans le modèle donné, l'état global du système vérifie à tout instant les propriétés de $\mathcal{P}_r$.

Un ensemble de valeurs d'entrée I représente les valeurs dont disposent chacun des processus du système $p_1, \ldots, p_n$ au commencement de l'algorithme. Il est possible qu'une partie des données soit partagée par plusieurs processus (voire tous), mais que d'autres valeurs initiales ne soient connues que d'un seul processus.

Un problème de *décision* est un problème pour lequel les processus doivent, pour le résoudre, retourner chacun une valeur de retour o_i, de telle sorte que l'ensemble $\mathcal{O}$ des valeurs de retour du système vérifie certaines propriétés. Dans un algorithme qui résout

un problème de décision, les processus qui retournent une valeur peuvent continuer à exécuter leur algorithme local, afin de communiquer des données aux autres processus, mais ne peuvent plus modifier leur valeur de retour.

2.3 Discussion sur le modèle asynchrone à pannes

Pour la suite de ce document, on ne considérera que des modèles *asynchrones* avec des *pannes* pour seules défaillances.

2.3.1 Quelques propriétés

Dans un tel modèle, que l'interface de communication soit par envoi de messages ou par mémoire partagée, il est impossible pour un processus p_i de détecter avec certitude une panne chez un autre processus p_j du système. En effet, même après une longue période sans communication par p_j visible du point de vue de p_i, il est possible que toutes ces communications soient arbitrairement retardées par l'asynchronisme du système, comme il est possible que p_j soit tombé en panne. Ainsi, une des difficultés majeures à étudier des problèmes et écrire des algorithmes dans ce modèle réside dans le fait que plusieurs situations possibles soient *indistinguables* par certains processus.

Cette propriété entraîne une technique de preuve de propriétés sur des algorithmes dans ce modèle. En effet, si le processus p_i exécute un algorithme déterministe, dans une situation donnée, il ne peut avoir qu'un seul comportement. Hors, si deux situations différentes sont indistinguables par p_i, ce dernier doit donc avoir le même comportement dans les deux cas. Ainsi, il suffit de trouver deux situations, atteignables dans l'exécution d'un algorithme, indistinguables par un processus, mais qui requièrent un comportement différent afin de correctement résoudre le problème, pour montrer que cet algorithme ne résout pas le problème en question. C'est ce type d'argument de preuve qui est notamment utilisé dans les preuves à arguments combinatoires et topologiques [20].

2.3.2 Pourquoi ces choix de modèle

Le choix du modèle par envoi de messages vient de la popularité de ce modèle (il constitue en quelque sorte la *base* de l'algorithmique distribuée), ainsi que de son réalisme vis-à-vis des systèmes distribués actuellement utilisés en pratique (bien qu'il soit simplifié par rapport aux détails techniques mis en place pour ces envois de messages).

L'asynchronisme permet d'avoir un modèle plus général, et représente bien les délais de communications variables, ainsi que les différences de vitesse de calcul des différentes machines pouvant faire partie d'un système distribué.

La choix des défaillances par pannes est lui aussi réaliste et très utilisé, tout en restant relativement général. Bien que seul un modèle byzantin puisse représenter la présence d'utilisateurs malveillants dans le système, les pannes suffisent à simuler la majorité des erreurs classiques qui peuvent survenir dans des systèmes réels.

L'hypothèse d'un graphe complet de canaux de communication sert de simplification pour ce modèle, et représente suffisamment bien la réalité, dans le sens où les messages d'un processus à un autre peuvent être transmis par un ou plusieurs processus intermédiaires en cas d'absence de canal direct. Tant que le graphe est suffisamment connecté pour éviter une déconnexion en cas de pannes trop nombreuses, les résultats de calculabilité sont équivalents.

L'utilisation de canaux de communication FIFO est réaliste, et permet l'écriture d'algorithmes plus simples. Enfin, la sûreté des canaux est un choix discutable, car des erreurs peuvent survenir dans des cas pratiques, mais cela évite d'avoir un modèle trop complexe,

l'association de l'asynchronisme et des pannes entraînant des algorithmes suffisamment difficiles à décrire et justifier. De plus, l'asynchronisme peut être vu comme un genre particulier de défaillances de canaux [5].

Ces hypothèses sur les canaux de communication complets, sûrs et FIFO, peuvent être considéré comme l'utilisation d'un modèle *en couches*. On suppose qu'une *couche* inférieure permet de garantir ces propriétés, et on en fait abstraction, afin de simplifier l'écriture d'algorithmes dans ce modèle.

2.4 Registres

Cette partie présente différents types de registres partagés connus.

2.4.1 Critère de cohérence et type de données

Outre les registres *atomiques*, il existe aussi deux autres types de registres avec des critères de cohérence plus faibles, pour le cas d'appels concurrents : les registres *sûrs* et les registres *réguliers*. Ces trois types de registres ont initialement été définis dans [68]. Pour des questions de simplicité de définition, ces types de registres sont considérés comme étant mono-écrivains, ce qui signifie que seul un processus peut appeler l'opération d'écriture, et donc que l'ensemble des opérations d'écriture lors d'une exécution est totalement ordonné. Pour tous ces types de registres, lorsqu'une opération de lecture n'est pas concurrente avec une opération d'écriture, elle retourne la valeur écrite lors de la dernière écriture qui la précède (ou $\perp$ si aucune écriture ne la précède).

Lors d'opérations concurrentes, un registre *sûr* ne vérifie pas de propriété supplémentaire, c'est-à-dire que toute lecture concurrente à une opération d'écriture peut retourner n'importe quelle valeur, y compris une qui n'a jamais été écrite. Un registre *régulier* vérifie la propriété suivante : une opération de lecture concurrente avec (au moins) une opération d'écriture peut retourner la valeur écrite lors de la dernière opération d'écriture précédant cette lecture (ou $\perp$ en absence d'une telle écriture), ou une des valeurs écrites lors d'une opération d'écriture concurrente avec cette lecture. La propriété du registre régulier est plus faible que la linéarisabilité, de part la possibilité d'avoir deux lectures successives (non concurrentes) dont la seconde retourne une valeur plus ancienne que la première, dans le cas où ces deux lectures sont concurrentes avec une même opération d'écriture.

Hors critère de cohérence, la notion de taille de registre est parfois considérée, généralement lors de l'utilisation de registres *binaires*, qui ne peuvent stocker que les valeurs 0 et 1. Lorsque rien n'est précisé à ce sujet, on suppose que les registres peuvent stocker des valeurs de taille arbitraire.

2.4.2 Nombre de lecteurs et d'écrivains

Comme évoqué précédemment, il est parfois question de registres *mono-écrivain* et *multi-lecteurs*, noté 1WnR ou encore SWMR, pour lesquels seul un processus (appelé processus *écrivain*) peut utiliser l'opération d'écriture, alors que tous les processus peuvent utiliser l'opération de lecture. Un registre général pour lequel chaque processus peut utiliser chacune des deux opérations est appelé *multi-écrivains* et *multi-lecteurs*, noté nWnR ou MWMR.

Il existe aussi des registres plus limités, appelés *mono-écrivain* et *mono-lecteur* (1W1R ou SWSR), pour lequel seul un processus écrivain peut appeler l'opération d'écriture, et un seul processus *lecteur* peut utiliser celle de lecture. Enfin, on peut aussi définir des registres *multi-écrivains* et *mono-lecteur*, pour lesquels tous les processus peuvent écrire, mais un seul processus peut appeler l'opération de lecture.

En pratique, on utilise plus souvent la version générale (nWnR), ou bien la version 1WnR dans un vecteur de registres. Ainsi, on peut disposer de n registres 1WnR, de telle sorte que chaque processus dispose d'un registre pour lequel il est le processus écrivain. Dans un tel contexte, il est possible d'utiliser des abstractions telles que le *snapshot*, et l'écriture de certains algorithmes y est simplifiée. Le *snapshot* est une opération permettant de lire l'état du vecteur de registres tel qu'il était à un instant durant cette opération, c'est-à-dire une lecture multi-registres vérifiant la linéarisabilité.

On peut aussi noter qu'il est possible de définir des registres intermédiaires disposant de plusieurs écrivains ou lecteurs, sans pour autant que tous les processus puissent écrire ou lire.

2.4.3 Registres à *test-and-set* et *compare-and-swap*

Certains objets partagés semblables aux registres à lecture-écriture disposent d'opérations plus puissantes, tout en fonctionnant de manière similaire, c'est-à-dire en stockant une valeur *écrite*, et en pouvant la retourner aux processus.

L'opération *test-and-set* prend en argument une valeur v à écrire, remplace la valeur stockée dans le registre par cette valeur v, et retourne la valeur v' qui était stockée dans le registre avant v. De plus, une telle opération est *atomique*, c'est-à-dire que d'un point de vue global et logique, tout se passe comme si cette opération entière était exécutée en un instant.

L'opération *compare-and-swap* prend en argument deux valeurs v_1 et v_2, et lit la valeur stockée dans le registre v, puis remplace cette valeur par v_1 si $v = v_2$. Ensuite, cette opération retourne la valeur v. Une fois de plus, cette opération est *atomique*.

2.4.4 Puissance relative des registres

Les registres à lecture-écriture sont tous équivalents, quel que soit leur type de données stockées (binaire ou valeurs arbitraire), leurs critères de cohérence (sûr, régulier, ou atomique) et leurs restrictions sur les lectures et écritures (de 1W1R à nWnR). En effet, il a été démontré (la chaîne de résultats successifs est résumée dans [21]) qu'un registre atomique nWnR stockant des valeurs arbitraires (c'est-à-dire le registre le plus *puissant* de toutes ces versions) pouvait être simulé à partir de registres binaires sûrs 1W1R (c'est-à-dire la version la moins puissante). Ainsi, en termes de calculabilité, tous ces modèles sont équivalents, d'où l'utilisation générale de registres atomiques stockant des valeurs arbitraires, afin de simplifier les raisonnements sans perdre en généralité.

Cependant, les registres disposant de l'opération *test-and-set* sont strictement plus puissants (en termes de calculabilité) que ceux à lecture-écriture, comme illustré dans [58], de part leur *consensus number* strictement supérieur. En effet, cette notion de *consensus number* introduite dans [58] permet de catégoriser la puissance d'objets de mémoire partagée, en indiquant le nombre maximal de processus n d'un système distribué pour lequel cet objet permet à ce système de résoudre le consensus. De même, les registres disposant de l'opération *compare-and-swap* sont strictement plus puissants que ceux disposant de *test-and-set*.

Chapitre 3

Analyse de l'existant

Ce chapitre présente des résultats connus dans le domaine.

La première partie détaille la simulation de mémoire partagée, ainsi que les implications de la *barrière* de la majorité de pannes. Cet algorithme est explicité pour plusieurs raisons : il s'agit d'un résultat classique et important qui permet de bien saisir la notion de barrière d'une majorité de pannes, il sert aussi de premier exemple d'algorithme distribué pour familiariser le lecteur avec ce type d'algorithme, et enfin les résultats du chapitre 4 s'en inspirent, d'où un intérêt à le présenter en amont.

La seconde partie présente quelques alternatives pour contourner cette barrière. Cela permet de comprendre quels travaux connexes existent, plus ou moins en rapport avec la barrière de la majorité de pannes. Cette partie aide aussi à définir le cadre de cette thèse et ses motivations.

Enfin, la troisième partie évoque quelques problèmes classiques de l'algorithmique distribuée. Ce tour d'horizon montre que plusieurs problèmes sont concernés par la barrière de la majorité de pannes, mais que ce n'est pas le cas de tous. Le problème du renommage, qui est au centre du chapitre 5, y est aussi présenté.

3.1 Simulation de mémoire partagée

3.1.1 La simulation *ABD* et les quorums

Comme évoqué dans le chapitre 1, il est possible de simuler des registres atomiques (mémoire partagée dans un système asynchrone par envoi de messages et avec des pannes. Ce résultat, désigné par l'acronyme *ABD* en référence à ses auteurs Attiya, Bar-Noy et Dolev [16], explicite une simulation de ces registres sous la condition qu'une stricte minorité des processus puisse tomber en panne($f < n/2$). De plus, il est prouvé qu'en présence d'une majorité de pannes ($f \geq n/2$), une telle simulation est impossible.

Le registre simulé dans cet algorithme est un registre atomique *mono-écrivain* et *multi-lecteurs* (abrégé en 1WnR), c'est-à-dire qu'un seul des processus du système, fixé pour ce registre, est susceptible d'appeler la fonction Ecriture. Ainsi, comme le processus *écrivain* ne peut appeler une opération qu'après avoir terminé la précédente, les opérations d'écritures (et les valeurs écrites associées) sont totalement ordonnées. Cela permet de reformuler la propriété de linéarisabilité plus simplement : Chaque opération de lecture retourne soit la valeur écrite par la dernière écriture *précédant* cette lecture, soit une valeur écrite par une écriture *concurrente* à cette lecture. De plus, si une lecture $\mathcal{L}_1$ retourne une valeur écrite pendant l'opération $\mathcal{E}_1$ et une autre lecture $\mathcal{L}_2$ retourne une valeur écrite pendant $\mathcal{E}_2$, et que $\mathcal{L}_1$ précède $\mathcal{L}_2$, alors $\mathcal{E}_2$ ne précède pas $\mathcal{E}_1$.

Le principe de la simulation est alors le suivant : Puisqu'une Lecture doit retourner la *dernière* valeur écrite, il est nécessaire de pouvoir ordonner les valeurs écrites. Un moyen

simple d'établir un ordre est que le processus *écrivain* stocke localement un compteur du nombre d'écritures effectuées ts, et l'incrémente à chaque nouvelle écriture pour ensuite l'associer à la valeur écrite. Chaque processus stocke ensuite localement une valeur qu'il suppose être la dernière valeur écrite, associée à l'*estampille* (*timestamp* en anglais) ts fournie par l'écrivain. Enfin, lorsque les processus s'échangent des messages, ils y joignent les données actuelles concernant ce qui devrait être la dernière valeur écrite, de manière à mettre à jour leurs variables locales en cas de réception de données plus récentes.

Ensuite, chaque opération requiert des communications entre les processus. Plus précisément, lors d'une Écriture, le processus écrivain envoie la nouvelle valeur (accompagnée de son estampille) à chacun des processus, puis attend des confirmations, jusqu'à savoir qu'au moins $n-f$ processus du système ont cette valeur. L'ensemble des processus ayant confirmé la réception de cette valeur est appelé un *quorum*, et contient généralement le processus ayant initié l'opération lui-même. Le principe d'une lecture est similaire, dans le sens où le processus qui appelle cette opération (on l'appellera ici *lecteur*) envoie à chaque processus un message, et attend d'obtenir des réponses de la part d'un quorum de $n - f$ processus (le lecteur compris). Ensuite, après avoir éventuellement mis à jour ses variables locales, il réitère ce procédé, envoyant un second message puis attendant des réponses d'un second quorum (éventuellement différent) de $n - f$ processus, afin de garantir l'atomicité du registre (c'est-à-dire la propriété de linéarisabilité des opérations), comme cela est démontré dans le lemme 3.1.8. En quelque sorte, ce second message permet d'*aider* l'écrivain à finir son écriture, en transmettant la valeur lue à un quorum. Enfin, le processus lecteur peut alors retourner la valeur qu'il a stocké localement, et qui a pu être mise à jour au cours de ces échanges.

Afin de garantir de bonnes communications, et pour ne pas considérer les réponses aux anciens messages (devenues obsolètes), un système de numéros de séquence *seq* est mis en place. À chaque fois qu'un processus p_i effectue une communication avec les processus du système, il incrémente sa variable locale seq_i, et la joint au message. Ainsi, les réponses à ce message peuvent aussi contenir cette même valeur seq_i, tandis que les réponses aux messages précédents contiennent des valeurs *seq* plus faibles, et peuvent donc être simplement ignorés.

Comme détaillé dans la partie 3.1.3, les propriétés vérifiées par cet algorithme sont basées sur l'intersection des quorums, qui est garantie par la stricte minorité de pannes.

3.1.2 Algorithme ABD et démonstrations

L'algorithme 3.1 est une retranscription de l'algorithme ABD présenté dans [16], légèrement reformulé afin de mieux correspondre aux notations de cette thèse. Sa présence permet d'illustrer la notion d'algorithme distribué dans le modèle asynchrone à pannes. De plus, les algorithmes présents dans le chapitre 4 s'inspirent largement de cet algorithme ABD, et sa compréhension permet de mieux appréhender les algorithmes à venir.

Algorithme 3.1 Simulation ABD de registre atomique (code exécuté par p_i)

1: **Initialisation**
2: $\quad\langle ts_i, v_i\rangle \leftarrow \langle 0, \perp\rangle$;
3: $\quad seq_i \leftarrow 0$;
4: **Fonction** LECTURE( )
5: $\quad$COMMUNIQUE(L_1, $\langle ts_i, v_i\rangle$) ;
6: $\quad\langle ts, v\rangle \leftarrow \langle ts_i, v_i\rangle$;
7: $\quad$COMMUNIQUE(L_2, $\langle ts, v\rangle$) ;
8: $\quad$**retourner** v ;
9: **Procédure** ECRITURE(v)

10: $\langle ts_e, v_e \rangle \leftarrow \langle ts_e + 1, v \rangle$;
11: COMMUNIQUE(E, $\langle ts_e, v_e \rangle$) ;
12: **Quand** un message m **est reçu** du processus p_j
13: **si** m est de type (L_1, seq) **alors**
14: **envoyer** une réponse $m' = (Re, seq, \langle ts_i, v_i \rangle)$ **à** p_j ;
15: **sinon si** m est de type (L_2, seq, $\langle ts, v \rangle$) ou (E, seq, $\langle ts, v \rangle$) **alors**
16: **si** $ts > ts_i$ **alors**
17: $\langle ts_i, v_i \rangle \leftarrow \langle ts, v \rangle$;
18: **envoyer** une réponse $m' = (Re, seq)$ **à** p_j ;
19: **Procédure** COMMUNIQUE($Type$, $\langle ts_0, v_0 \rangle$)
20: $seq_i \leftarrow seq_i + 1$;
21: $Reponses \leftarrow \emptyset$;
22: **si** $Type = L_1$ **alors**
23: **pour chaque** processus p du système **faire**
24: **envoyer** le message $m = (Type, seq_i)$ **à** p ;
25: **tant que** $|Reponses| < n - f$ **faire**
26: **attendre** de recevoir un message (Re, seq_i, $\langle ts, v \rangle$) d'un processus p_j ;
27: $Reponses \leftarrow Reponses \cup \{p_j\}$;
28: **si** $ts > ts_i$ **alors**
29: $\langle ts_i, v_i \rangle \leftarrow \langle ts, v \rangle$;
30: **sinon si** $Type = L_2$ ou $Type = E$ **alors**
31: **pour chaque** processus p du système **faire**
32: **envoyer** le message $m = (Type, seq_i, \langle ts_0, v_0 \rangle)$ **à** p ;
33: **tant que** $|Reponses| < n - f$ **faire**
34: **attendre** de recevoir un message (Re, seq_i) d'un processus p_j ;
35: $Reponses \leftarrow Reponses \cup \{p_j\}$;

Les preuves suivantes que cet algorithme vérifie bien les propriétés attendues sont fortement inspirées de celles présentes dans [16].

Lemme 3.1.1. *Si un processus p_i* correct *appelle la fonction communique(), alors cet appel termine.*

Démonstration. Le processus p_i commence par envoyer un message à chacun des processus du système (ligne 24 et 32). Chacun de ces messages sera reçu sauf si son destinataire tombe en panne, car les canaux de communication sont *sûrs*. Une fois un tel message reçu, le processus destinataire p_j envoie une réponse de type Re, quel que soit le type de message utilisé dans la fonction Communique (lignes 14 ou 18). Comme p_i est correct, chacune de ces réponses sera reçue au bout d'un certain temps.

Le système comprend au plus f processus susceptibles de tomber en panne au cours d'une exécution, donc au moins $n - f$ d'entre eux ne tomberont pas en panne. Ainsi, le processus p_i recevra des réponses d'au moins $n-f$ processus différents. De plus, le numéro de séquence seq_i initialement transmis dans le message envoyé par p_i est recopié dans la réponse Re. Donc au bout d'un certain temps $|Reponses| \geq n - f$ (ligne 25 et 33), et donc Communique() termine. □

Corollaire 3.1.2. *Chaque opération de Lecture ou d'Écriture par un processus correct termine.*

Lemme 3.1.3. *Lors de n'importe quelle exécution, les valeurs successives de la variable locale ts_i du processus p_i sont croissantes.*

Démonstration. Les seuls moments où cette variable est modifiée sont aux lignes 10, 17, et 29, qui vérifient bien que la nouvelle valeur est plus grande que la précédente. □

Lemme 3.1.4. *Lorsqu'une exécution C de la fonction Communique() de type L_2 ou E avec $\langle ts_0, v_0\rangle$ pour paramètre termine, au moins $n-f$ processus vérifient $ts_i \geq ts_0$, où les ts_i sont les variables locales de ces processus.*

Démonstration. A la fin de C, la variable *Reponses* contient au moins $n-f$ processus tels que chacun de ces processus a reçu le message envoyé au début de C (ligne 32) et y a répondu par un message de type Re. En effet, le numéro de séquence seq_j utilisé permet d'identifier la réponse Re comme correspondant bien à cette exécution de la fonction Communique(). De plus, lors de la réception de ce message contenant $\langle ts_0, v_0\rangle$, un processus p_i stocke ts_0 dans sa variable ts_i si $ts_0 > ts_i$ (ligne 17). Enfin, les valeurs successives des variables ts_i sont croissantes. Donc, chaque processus p_i qui reçoit le message vérifie $ts_i \geq ts_0$. Cela signifie que tous les processus de l'ensemble *Reponses* vérifient cette propriété, ce qui prouve ce lemme. □

Lemme 3.1.5. *Lorsqu'une exécution C de la fonction Communique() de type L_1 par un processus p_j termine, au moins $n-f$ processus vérifient $ts_j \geq ts_i^0$, où les ts_i^0 sont les valeurs des variables locales de ces processus lors de l'appel à cette fonction Communique().*

Démonstration. A la fin de C, la variable *Reponses* contient au moins $n-f$ processus tels que chacun de ces processus a reçu le message envoyé au début de C (ligne 24) et y a répondu par un message de type Re. En effet, le numéro de séquence seq_j utilisé permet d'identifier la réponse Re comme correspondant bien à cette exécution de la fonction Communique(). Chacune de ces réponses contient une valeur ts_i^1, qui est la valeur de ts_i au moment où p_i a envoyé cette réponse (ligne 14). Comme les valeurs stockées dans les variables ts_i sont croissantes, $ts_i^0 \leq ts_i^1$. Lorsque le processus p_j reçoit une telle réponse, si $ts_j < ts_i^1$ alors ts_j prend pour nouvelle valeur ts_i^1 (ligne 29). En particulier, à la fin de C, on a $ts_j \geq ts_i^1$ pour chaque ts_i^1 reçu par un message d'un des processus de *Reponses*. Donc, pour chaque processus p_i dans *Reponses*, on a $ts_j \geq ts_i^0$, ce qui prouve le lemme. □

Corollaire 3.1.6. *Supposons qu'une exécution C de la fonction Communique() de type L_2 ou E et avec $\langle ts_0, v_0\rangle$ pour paramètre termine avant qu'une autre exécution C' de Communique de type L_1 ne soit appelée par un processus correct p_j. Alors, à la fin de l'exécution C', on a $ts_j \geq ts_0$, où ts_j est la variable locale de p_j.*

Démonstration. On note R l'ensemble *Reponses* de C et R' celui de C'. D'après les deux lemmes précédents, on a, à la fin de C, $\forall p_i \in R$, $ts_i \geq ts_0$, et à la fin de C', $\forall p_i \in R'$, $ts_j \geq ts_i^0$. Or, comme C termine avant que C' ne commence, on a $\forall p_i \in R \cap R'$, $ts_0 \leq ts_i^0$. Donc, si $R \cap R' \neq \emptyset$, à la fin de C', $ts_j \geq ts_0$ par transitivité. Hors, on sait que $|R| \geq n-f$, $|R'| \geq n-f$, $R \subseteq \mathcal{P}$, $R' \subseteq \mathcal{P}$, et $f < n/2$, avec $|\mathcal{P}| = n$. Donc, $2 * (n-f) > n$, ce qui signifie que $R \cap R' \neq \emptyset$, ce qui prouve ce corollaire. □

Lemme 3.1.7. *Une opération de lecture retourne la valeur écrite par la dernière opération d'écriture précédant cette lecture, ou une valeur écrite par une opération d'écriture concurrente.*

Démonstration. D'après l'algorithme, si une opération d'écriture par p_e *précède* une opération de lecture par p_j, alors un appel à la fonction Communique() de type E avec pour paramètre le $\langle ts_0, v_0\rangle$ correspondant à l'écriture en question a terminé avant que Communique() de type L_1 ne soit appelé par p_j. Donc, d'après le corollaire précédent, à la fin de ce Communique() de type L_1, $ts_j \geq ts_0$. Cela signifie que la valeur v retournée la lecture est associée à une estampille ts vérifiant $ts \geq ts_0$. Or, les valeurs de ts sont croissantes, et

associées de manière unique aux valeurs v par le processus écrivain p_e lors des opérations d'écriture (ligne 10). Ainsi, la valeur retournée par la lecture est au moins aussi récente que celle écrite lors de la dernière écriture précédent cette lecture, ce qui prouve ce lemme. □

Lemme 3.1.8. *Si une lecture $\mathcal{L}_1$ retourne une valeur écrite pendant l'opération $\mathcal{E}_1$ et une autre lecture $\mathcal{L}_2$ retourne une valeur écrite pendant $\mathcal{E}_2$, et que $\mathcal{L}_1$ précède $\mathcal{L}_2$, alors $\mathcal{E}_2$ ne précède pas $\mathcal{E}_1$.*

Démonstration. Si $\mathcal{L}_1$ précède $\mathcal{L}_2$, cela signifie que Communique() de type L_2 a été appelé (au cours de $\mathcal{L}_1$) avec pour paramètre $\langle ts, v\rangle$ et a terminé avant que Communique() de type L_1 ne soit appelé dans le cadre de l'opération $\mathcal{L}_2$. Et $\langle ts, v\rangle$ est ici la valeur retournée par la lecture $\mathcal{L}_1$. En particulier, par le corollaire précédent, on a $ts' \geq ts$ où $\langle ts', v'\rangle$ est la valeur retournée par la lecture $\mathcal{L}_2$. Or, comme les valeurs de ts sont croissantes et associées de manière unique aux valeurs v par le processus écrivain p_e, on sait que l'écriture $\mathcal{E}_2$ ne peut pas précéder l'écriture $\mathcal{E}_1$. □

Propriété 3.1.9. *L'algorithme précédent simule un registre atomique 1WnR en présence d'une stricte minorité de pannes.*

3.1.3 Barrière de la majorité de pannes et théorème CAP

Dans la simulation précédente, l'utilisation de quorums de $n - f$ processus est le point clé permettant de vérifier les propriétés du registre. Comme au plus f des n processus du système sont susceptibles de tomber en panne, au moins $n - f$ processus sont assurés de ne pas être en panne, à tout instant. Donc, en attendant des réponses de la part de $n - f$ processus (lui-même inclus), le processus initiant l'opération est certain d'obtenir une réponse au bout d'un certain temps, car chaque processus qui ne tombe pas en panne doit répondre lorsqu'il reçoit un message de ce type, et cela arrive grâce à la sûreté des canaux de communication.

De plus, pour préserver la propriété de cohérence des données (chaque lecture retourne la dernière valeur écrite), il faut que ces données soient transmises du processus écrivain vers les processus lecteurs. En d'autres termes, comme le processus écrivain sait qu'au moins un quorum de $n - f$ processus disposent de cette valeur, il faut ensuite que le processus lecteur obtienne un message de la part d'un de ces processus afin de garantir des valeurs suffisamment récentes. Comme le processus lecteur communique lui aussi avec un (et même deux) quorum(s) de $n-f$ processus, une telle garantie arrive si ce quorum a une intersection non-vide avec le quorum de l'écrivain. Puisque les processus ne choisissent pas quels sont les processus de leurs quorums pour ces opérations, il suffit que chaque quorum contienne strictement plus de $n/2$ processus pour garantir qu'au moins un processus fasse partie des 2 quorums. Ainsi, si $n - f > n/2$, autrement dit si $f < n/2$, alors la cohérence des données est bien préservée. Ce système de quorums à intersection non-vide est si pratique que des abstractions supposent parfois avoir un système l'utilisant, bien que les circonstances soient différentes, comme dans [48].

D'autre part, en présence d'une majorité de pannes ($f \geq n/2$), le système se retrouve partitionné, et on ne peut alors plus garantir à la fois la terminaison des opérations (disponibilité) et la cohérence des données, d'après le théorème CAP introduit dans le chapitre 1. En effet, il est possible que, lors d'une exécution particulière, le système soit séparé en 2 parties de $n/2$ processus chacune telle que les messages d'un processus d'une partie vers un processus de l'autre partie soient tous arbitrairement ralentis, alors que les messages internes à une partie arrivent rapidement. Dans ces conditions, si on souhaite garantir la terminaison des opérations, il est nécessaire que les processus n'attendent pas de réponses de plus de $n - f$ processus, à cause de la possibilité que f processus

soient tombés en panne (ce qui impliquerait un risque d'attendre indéfiniment). Ainsi, les opérations dans une partie du système n'attendent pas de réponses de la part des processus de l'autre partie, à cause de l'indistinguabilité avec le scénario dans lequel tous les processus de cette autre partie sont en panne. Dans le cas où chaque processus n'attend des réponses que d'au plus $n-f$ processus, il est donc possible qu'une opération de lecture termine dans une partie, retournant la valeur la plus récente connue des processus de cette partie, alors que des valeurs plus récentes ont déjà été préalablement écrites dans l'autre partie. Cette contradiction avec la règle de cohérence qui demande aux lectures de retourner la dernière valeur écrite montre bien qu'une telle simulation de registres atomiques est impossible en présence d'une majorité de pannes.

Enfin, de nombreux problèmes, dans notre modèle asynchrone par envoi de messages et avec pannes, ont aussi une limitation similaire à une minorité de pannes, car des arguments semblables à ceux présentés ici peuvent généralement être appliqués pour prouver l'impossibilité de garantir à la fois la disponibilité et la cohérence des données en présence d'une majorité de pannes, synonyme de partitionnement. Le théorème CAP représente bien la limite entre l'utilisation de quorums de taille au plus $n - f$ pour garantir la disponibilité, et celle de quorums de taille plus de $n/2$ pour garantir la cohérence.

Les systèmes utilisant cette notion de *quorums* pour garantir une transmission de données n'est pas nouvelle [26, 81, 74], et est utilisée depuis longtemps, notamment pour résoudre des problèmes d'exclusion mutuelle [85].

3.2 Ce qui est fait pour contourner la barrière

Depuis ce théorème CAP (et aussi avant, dans une moindre mesure), de nombreux travaux ont été effectués pour chercher passer outre cette limitation, comme cela est résumé dans [47, 55] entre autre.

3.2.1 Approche probabiliste

Une idée pour contourner cette barrière d'une majorité de pannes est d'utiliser la notion de *quorums probabilistes* [75, 63]. Dans ce contexte, on suppose que chaque processus client communique lors d'une opération avec un (ou plusieurs) ensemble de processus appelé *quorum*. De plus, contrairement au cas décrit précédemment (dans l'algorithme ABD) où un tel quorum n'est choisi que par la vitesse des réponses obtenues, et donc par un adversaire arbitraire qu'est l'asynchronisme, les quorums probabilistes sont choisis aléatoirement selon une certaine distribution de probabilité (ou *stratégie*) ω. On a alors un système de quorum à ε-intersection [75] si 2 quorums Q_1 et Q_2 choisis aléatoirement selon ω ont une probabilité $1-\varepsilon$ de vérifier $Q_1 \cap Q_2 \neq \emptyset$. En d'autres termes, une cohérence des données est vérifiée avec forte probabilité si ε est faible, mais les incohérences restent possibles. Autrement dit, chaque lecture a une forte probabilité de retourner la dernière valeur écrite. De plus, un tel système utilise une notion de *probabilité d'échec* pour représenter la possibilité de ne pas pouvoir terminer une opération si chaque quorum possible comprend un processus en panne. Cette notion est aussi présente dans les systèmes à quorums *stricts* pour lesquels la probabilité d'échec tend vers 0 lorsque $f < n/2$ mais tend vers 1 lorsque $f > n/2$, ce qui souligne encore une fois la *barrière* de la majorité de pannes.

D'autres systèmes, comme dans [10], supposent que les pannes sont aléatoires, et que les processus en panne peuvent se rétablir au bout d'un certain temps. Dans un tel contexte, il est possible de développer des systèmes à *k-obsolescence bornée* (en anglais *k-bounded staleness*), qui limitent les valeurs lues dans le système (de manière similaire aux registres à lecture/écriture) à l'une des k dernières valeurs écrites (au lieu de *la* dernière valeur écrite). Ainsi, la propriété de linéarisabilité est toujours vérifiée, dans le

sens où chaque opération est *atomique* (tout se passe comme si elle ne prenait qu'un instant), mais la spécification autorise chaque lecture à retourner une valeur écrite par l'une des k dernières écritures précédant cette lecture. L'utilisation de cette notion de défaillances probabilistes et temporaires permet ainsi de contourner les problèmes de cohérence habituels. Et l'idée d'obsolescence bornée est par ailleurs intéressante et utilisée dans d'autres travaux, comme [11] par exemple. Des extensions de registre vérifiant la propriété de k-obsolescence bornée sont d'ailleurs équivalent aux registres habituels [91]. En effet, il est possible à partir de registres à k-obsolescence bornée de simuler des registres atomiques classiques : ces objets ont donc la même *puissance* en termes de calculabilité.

Dans [24], l'idée de quorums probabilistes est rapprochée de celle de Cohérence finale. La notion de *cohérence finale* signifie que, si lors d'une exécution, à partir d'un certain temps t, plus aucune opération d'écriture n'est exécutée, alors à partir d'un certain temps t', toutes les lectures retournent la même valeur (le système se stabilise). Afin de vérifier de meilleures propriétés, le concept d'*obsolescence bornée probabiliste* est développé, limitant les valeurs lues dans un système (à la manière des registres à lecture/écriture) à retourner une des k dernières valeurs écrites avec probabilité $1 - \varepsilon$. De cette manière, cela renforce la notion de cohérence finale en garantissant une forte probabilité de cohérence *suffisante*, retournant une valeur récente (une des k dernières) sans nécessairement être la dernière valeur écrite. Enfin, dans [24], une étude des systèmes utilisés en pratique montre que l'*âge* (c'est-à-dire le nombre d'écriture qui ont eu lieu entre une lecture et l'écriture correspondant à la valeur retournée par cette lecture) des données lues est faible, et que l'utilisation de quorums probabilistes permet ainsi de gagner en latence (en utilisant des quorums de plus petite taille), avec une perte de cohérence négligeable.

Toutes ces notions probabilistes sont intéressantes pour des applications pratiques, car la faible probabilité d'échec est souvent acceptable au vu des gains en *disponibilité* et de la plus grande tolérance aux pannes et partitionnement. Néanmoins, de tels résultats ne pourraient pas être utilisés dans des systèmes plus critiques, où même une faible probabilité est trop élevée et le risque se doit d'être réduit à zéro. Ainsi, cette thèse s'inscrit dans une étude déterministe, afin d'établir de véritables garanties non-probabilistes.

3.2.2 Cohérence finale

Comme évoqué dans le chapitre 1, la *cohérence finale* (en anglais *eventual consistency*) représente l'idée de ne pas garantir une cohérence parfaite, au profit de la disponibilité et de la résistance au partitionnement, mais de garantir une certaine *progression* dans la transmission des données, de sorte qu'en cas d'absence d'écriture, le système se stabilise, chaque processus étant mis à jour au bout d'un moment. Bien que cette notion n'apporte en soi que peu de garanties de cohérence, elle est souvent utilisée sous une forme ou une autre en pratique [28, 45, 22], de par ses bonnes propriétés dans un contexte favorable tout en gardant une disponibilité en cas de défaillances. C'est par exemple le cas évoqué dans la partie précédente, où l'utilisation d'algorithmes probabilistes permet de limiter les risques d'obsolescence des données à une faible probabilité.

Un exemple significatif de système utilisant ce principe est celui de Dynamo, un système de stockage de données utilisé par Amazon.com [41]. Ce système utilise des équivalents de registres à lecture/écriture (les *key-value store*), dont les données sont répliquées sur plusieurs processus (à la manière de l'implémentation ABD des registres atomiques). Lorsque des défaillances entraînent un partitionnement, les quorums utilisés peuvent ne pas s'intersecter, générant ainsi des incohérences des données. Mais un système d'échange régulier de données permet au bout d'un certain temps de corriger ces erreurs, et la pratique montre que ces erreurs ne sont généralement pas critiques, et corrigées en un temps satisfaisant [27]. Cassandra [65] est un autre système, utilisé par Facebook, qui

est basé sur le même principe de cohérence finale au profit d'une meilleure disponibilité.

En plus de la notion de cohérence finale, on souhaite généralement vérifier d'autres propriétés limitant les incohérences, comme décrit dans [92]. On y retrouve des propriétés *locales* empêchant des incohérences flagrantes entre différentes opérations effectuées par un même processus client. Par exemple, si un processus exécute une opération de type écriture, puis une seconde de type lecture, il semble utile d'imposer que les valeurs lues contiennent celles précédemment écrites, ou des valeurs plus récentes. De même, si un processus client effectue plusieurs lectures de données, on s'attend à ce que chaque opération retourne des valeurs au moins aussi récentes que la précédente. Des garanties globales plus fortes sont souhaitables, mais pas toujours possibles, comme la propriété d'*obsolescence bornée*, qui ne peut être assurée de manière déterministe dans un modèle avec une majorité de pannes arbitraires.

Enfin, un autre point étudié en rapport avec la cohérence finale est celui de la gestion de conflits, qui peuvent par exemple survenir si plusieurs opérations de type écriture sont réalisés concurremment sur des mêmes données. Par exemple, l'utilisation de *vecteurs de versions* [84] permet d'établir un ordre partiel sur les différentes données, et simplifie la résolution des conflits, qui, quant à elle, varie selon les systèmes précisément utilisés. D'autres systèmes valorisent l'utilisation d'opérations *commutatives*, afin de garantir une convergence vers une cohérence finale, en limitant les contraintes sur l'ordre de réception des différents messages entre les processus, comme dans par exemple [87] et [39]. Il est aussi possible de considérer plusieurs versions comme correctes, en perdant potentiellement la cohérence finale [35].

En somme, la cohérence finale est un moyen de contourner la barrière de la majorité de pannes, afin d'améliorer les performances en pratique de larges systèmes distribués. Cette approche basée principalement sur des problèmes applicatifs (avec des évolutions, comme souligné par [57]) diffère en cela de celle de cette thèse, qui cherche à approfondir l'étude de problèmes fondamentaux de l'algorithmique distribuée, sans application pratique *directe*.

3.2.3 Cohérence causale et cohérence pipeline

La cohérence *pipeline* (issue de [72]) et la cohérence *causale* [9] sont deux autres formes de cohérences *faibles*, qui diffèrent de la cohérence finale. Comme souligné dans [47], ces critères ne sont pas directement comparables avec la cohérence finale, car ils ne sont ni plus forts ni plus faibles de cette dernière (dans le sens où la vérification d'une propriété d'un de ces types de cohérence n'implique pas l'autre). Cependant, la cohérence pipeline est strictement plus faible que la cohérence causale, et est relativement peu utilisée.

La cohérence causale [9] demande à ce que plusieurs opérations exécutées par un même processus (l'une après l'autre) soient globalement perçues (du point de vue des autres processus) comme ayant eu lieu dans le même ordre que ce qu'ils ont réellement eu lieu. Ainsi, cela peut être fait à l'aide d'estampilles, à la façon de la simulation ABD. De plus, si un processus effectue une opération de type lecture et obtient ainsi des données produites par une opération de type écriture (pouvant avoir été exécutée par un autre processus), cette opération de lecture est globalement (et logiquement) vue comme postérieure à l'opération d'écriture correspondante, du point de vue de tout processus au courant de l'existence de ces opérations.

Ce critère est renforcé en cohérence *causale+* dans [73], en ajoutant une gestion des conflits, afin d'aider à converger vers une possible version finale. Plus précisément, il s'agit de cohérence causale étendue avec de la cohérence finale. Inversement, [23] propose l'ajout de cohérence causale sur des système vérifiant la cohérence finale. De telles combinaisons sont utiles en pratique, et notamment visibles dans [12], qui présente un système (key-value store) géo-distribué vérifiant la cohérence causale+.

L'étude de la cohérence causale est cependant moins intéressante que celle de cohérence finale. En effet, celle-ci n'est bien définie que pour des objets de type registre (comme souligné dans [47]), et est moins généralisable que la cohérence finale. D'autre part, il est possible d'ajouter automatiquement la cohérence causale à un système vérifiant la cohérence finale, comme cela est décrit dans [23]. Enfin, un tel critère de cohérence n'a de réel intérêt que lors d'utilisation d'objets multi-écrivains ou de plusieurs objets mono-écrivains ayant des écrivains différents, et l'étude de cette thèse porte principalement sur des objets mono-écrivains isolés.

3.3 Divers problèmes et résultats classiques

Cette partie présente trois problèmes classiques et importants du domaine que l'on considère. On y découvre par exemple que le consensus et le k-accord ensembliste ne sont pas concernés par la barrière de la majorité de pannes, contrairement au renommage et à l'accord approximatif. De plus, le renommage est ici présenté en amont du chapitre 5, dans lequel il sera étudié plus en profondeur.

3.3.1 Consensus et k-accord ensembliste

Le *consensus*, ainsi que son extension le *k-accord ensembliste* (*k-set agreement* en anglais), est un problème classique et important dans le monde de l'algorithmique distribuée. En effet, il est possible, si on est capable de résoudre le consensus, de simuler n'importe quel objet partagé, comme décrit par la notion d'*universalité* du consensus décrit dans [58, 50]. Plus généralement, le consensus permet la synchronisation de *machines à état* qui peuvent être vues comme la base de la programmation distribuée [66].

Le consensus est défini comme suit : Chaque processus reçoit une valeur initiale parmi une des valeurs initiales possibles V, formant ainsi un ensemble des valeurs initiales pour une exécution donnée $\mathcal{I} \subseteq V$. Ensuite, chaque processus qui ne tombe pas en panne doit retourner une valeur de sortie, en un temps fini, de telle sorte que l'ensemble des valeurs de sortie de tous les processus du système, $\mathcal{O}$, ne contienne qu'une unique valeur $\mathcal{O} = \{v_o\}$ et que cette valeur fasse partie des valeurs initiales du système pour cette exécution : $v_o \in \mathcal{I}$.

Dans le modèle que l'on considère, à savoir le modèle asynchrone à pannes, il a été montré [46] que le consensus ne peut pas être résolu dès lors qu'au moins un processus est susceptible de tomber en panne ($f \geq 1$), et ce même dans le cas où l'ensemble des valeurs initiales possibles (V) est restreint à l'ensemble $\{0, 1\}$.

Cette limite franche a entraîné une extension assez naturelle du problème du consensus en celui du k-accord ensembliste, défini comme suit : Chaque processus reçoit une valeur initiale de la même manière que pour le consensus. Chaque processus qui ne tombe pas en panne doit retourner une valeur, de manière à ce que l'ensemble des valeurs de retour $\mathcal{O}$ contienne au plus k valeurs différentes ($|\mathcal{O}| \leq k$) et que ces valeurs soient toutes dans l'ensemble des valeurs initiales de cette exécution : $\mathcal{O} \subseteq \mathcal{I}$.

Ainsi, on constate que dans le cas où $k = 1$, le k-accord ensembliste est exactement le consensus. Ce problème a lui aussi été étudié dans le modèle qui nous intéresse, pour lequel il a été prouvé [31] qu'il pouvait être résolu si et seulement si le nombre de pannes possibles f est strictement inférieur à k, le nombre de valeurs de retour différentes possibles. D'autres résultats étudient les critères minimaux que doivent vérifier le système pour que ce problème puisse être résolu [34].

Enfin, ce problème est intéressant de part sa simplicité, car il est alors possible d'utiliser ce résultat pour montrer des impossibilités par réduction au problème du k-accord ensembliste.

3.3.2 Renommage et diviseurs

Un autre problème classique, et pour lequel nous entrerons dans plus de détails dans le chapitre 5, est le problème du *renommage*. Comme évoqué dans le chapitre 1, et comme cela sera rappelé dans le chapitre 5, il s'agit d'un problème intéressant en soi d'un point de vue théorique et largement étudié, qui a de plus des applications pratiques par exemple dans l'allocation de ressources.

Ce problème est défini comme suit : Chaque processus dispose d'un identifiant unique (comme précisé dans la 1ère partie de ce chapitre). Chaque processus p_i qui ne tombe pas en panne doit retourner, en un temps fini, un entier v_i, de telle sorte que chaque valeur de retour soit unique : $\forall i \neq j, v_i \neq v_j$. Enfin, un algorithme de renommage a un *espace de noms* de taille M si, pour toute exécution possible de cet algorithme, l'ensemble des valeurs de retour ne contient que des entiers entre 1 et M : $\mathcal{O} \subseteq [1..M]$.

Qualitativement, les algorithmes de renommage ont pour objectif de résoudre le renommage avec un espace de noms aussi petit que possible, tout en gardant des notions de rapidité d'exécution. Résoudre ce problème de renommage peut avoir des applications dans des domaines tels que l'allocation de multiples ressources (en lien avec l'*exclusion mutuelle*), ou peut encore servir à remplacer l'identifiant unique de chaque processus dans le but d'exécuter plus rapidement un algorithme qui dépend de ces identifiants.

Historiquement, le problème du renommage a été introduit [17] dans les débuts de l'étude du modèle asynchrone par envoi de messages et à pannes, afin d'amener un problème non-trivial et intéressant mais pouvant être résolu dans ce modèle. Par la suite, ce problème a été étudié dans de nombreux articles [37, 82, 78, 32, 79], mais principalement dans un modèle asynchrone à mémoire partagée (plus précisément à l'aide de registres atomiques) et avec pannes. Cependant, grâce à la simulation de registres atomiques introduite précédemment, les algorithmes de renommage dans ce modèle à mémoire partagée peuvent aussi être appliqués au cas par envoi de messages.

Un de ces résultats [78] est notamment intéressant pour son introduction d'une variante du problème classique du renommage dit *à-un-coup* (*one-shot* en anglais), qui est le renommage *réutilisable* (*long-lived* en anglais). Dans cet objectif, les auteurs introduisent 2 versions d'un objet particulier simple mais intéressant, plus tard appelé *diviseur* (*splitter* en anglais).

Les diviseurs sont des objets qui peuvent être simulés à partir de registres atomiques, et donc indirectement dans le modèle par envoi de messages. Ils sont accessibles via une unique opération qui vérifie les propriétés suivantes : Chaque appel par un processus correct termine, et retourne une direction parmi $\{droite, bas, stop\}$. Au plus un appel peut retourner *stop* au cours d'une exécution. Tous les appels ne retournent pas *bas*, et tous les appels ne retournent pas *droite*.

Malgré cette définition relativement simple, les diviseurs permettent de résoudre le renommage de manière efficace et simple [78]. Mais, comme leur simulation est basée sur celle des registres atomiques, ils ne peuvent exister dans le modèle par envoi de message qu'en présence d'une stricte minorité de pannes. De manière similaire, le problème du renommage ne peut lui non plus pas être résolu en présence d'une majorité de pannes, et se voit contraint au modèle par envoi de messages avec un stricte minorité de pannes.

3.3.3 Accord approximatif

Le problème de l'*accord approximatif* (*approximate agreement* en anglais), aussi appelé *accord-ε* peut être considéré comme une variante du problème du consensus [44, 43]. Contrairement au consensus, ce problème se restreint cependant à des nombres réels plutôt qu'à des valeurs arbitraires, bien qu'il puisse être étendu à tout espace doté d'une métrique.

Il est défini comme suit : Chaque processus reçoit une valeur initiale parmi une des valeurs initiales possibles $V \subseteq \mathbb{R}$, formant ainsi un ensemble des valeurs initiales pour une exécution donnée $\mathcal{I} \subseteq V$. Chaque processus obtient aussi la valeur commune $\varepsilon > 0$, ainsi que 2 bornes a et b telles que $\mathcal{I} \subseteq [a, b]$. Ensuite, chaque processus qui ne tombe pas en panne doit retourner une valeur de sortie, en un temps fini, de telle sorte que l'ensemble des valeurs de sortie de tous les processus du système, $\mathcal{O}$, ne contienne que des valeurs proches à ε près : $\forall u, v \in \mathcal{O}$, $|u - v| \leq \varepsilon$. De plus, ces valeurs doivent être contenues dans l'intervalle formé par les valeurs initiales : $\mathcal{O} \subseteq [min(\mathcal{I}), max(\mathcal{I})]$.

Ce problème peut aussi être défini comme un problème qui n'est pas un problème de décision, et pour lequel les processus n'ont pas besoin de connaître ε et les bornes a et b. Dans cette version, chaque processus stocke localement une valeur v initialisé par sa valeur de départ, et qu'il va faire évoluer au cours de l'algorithme. L'algorithme pour chaque processus est alors divisé en *rondes* (ou *rounds*), et doit vérifier que, pour chaque ε et ensemble de valeurs d'entrée, il existe une ronde r_0 à partir de laquelle les valeurs stockées par des processus qui ne sont pas en panne soient toutes proches à ε près. Autrement dit, à tout instant, si chaque processus (qui n'est pas en panne) est dans son algorithme à une ronde supérieure à r_0, alors leurs valeurs stockées sont telles que $\forall v_i, v_j$, $|v_i - v_j| \leq \varepsilon$ et $v_i \in [min(\mathcal{I}), max(\mathcal{I})]$. Il est à noter que, sans connaissance de bornes a et b sur $\mathcal{I}$ et de ε, les processus ne peuvent pas connaître r_0. D'autres variations du problème sont aussi possibles, comme dans [76].

Quelle que soit la définition retenue, le problème de l'accord approximatif peut être résolu dans le modèle asynchrone par envoi de messages avec une stricte minorité de pannes [44]. Cependant, il ne peut pas être résolu en présence d'une majorité de pannes. En effet, si le système est séparé en 2 parties, de telle sorte que les messages entre 2 parties différentes soient arbitrairement ralentis, chaque processus d'une partie va penser que les processus de l'autre partie sont tombés en panne. Ainsi, si tous les processus d'une partie ont pour valeur initiale x, et tous les processus de l'autre partie ont pour valeur initiale y, chaque processus va garder sa valeur pendant un nombre arbitraire de rondes, ou la retourner, selon la version du problème. De cette manière, si $|x - y| > \varepsilon$, il est impossible de vérifier les propriétés souhaitées.

Chapitre 4

Extension des registres partagés

Ce chapitre traite de l'extension des registres partagés habituels (présentés en partie 2.4), dans le but de les implémenter dans un modèle par envoi de message en présence d'une majorité de pannes.

On définit tout d'abord les registres χ-colorés, qui sont des registres pour lesquels la propriété de disponibilité est affaiblie. Ces registres peuvent être implémentés avec $\chi = 2f - n + 2$.

Ensuite, on définit les α-registres, dont la propriété de cohérence est affaiblie. Un algorithme qui les implémente avec $\alpha = 2(2f - n + 2) - 1$ est présenté. Puis une borne inférieure démontre qu'il est impossible d'implémenter de tels registres avec $\alpha < 2f - n + 2 + 1$. Enfin, on étudie la composition de plusieurs α-registres mono-écrivains et l'impossibilité de tels registres multi-écrivains.

4.1 Affaiblissement de la disponibilité : les registres χ-colorés

4.1.1 Définition

Comme cela a déjà été expliqué dans les chapitres précédents, il est impossible de simuler des registres partagés *classiques* en présence d'une majorité de pannes. Pour simuler des objets semblables aux registres habituels dans un tel contexte, il faut affaiblir une des propriétés usuelles. Une des possibilités est donc de relâcher la *disponibilité*, en permettant à certains registres de ne jamais terminer certaines opérations. Afin d'éviter un blocage total du système, on peut alors utiliser plusieurs pseudo-registres en parallèle, car il est alors possible de garantir une certaine propriété de disponibilité sur l'ensemble de ces objets partagés. On ne considère pour l'instant que des simili-registres qui sont mono-écrivain multi-lecteurs (1WnR). Ainsi, on définit un *banc* de registres χ-colorés (aussi appelé *banc* de χ registres colorés) comme un ensemble ordonné de χ objets partagés disposant chacun des opérations Lecture et Ecriture, et vérifiant les propriétés suivantes :

- *mono-écrivain* : Un seul processus p_e peut appeler l'opération d'écriture sur chacun des χ registres de ce banc.
- *linéarisabilité* : Chaque registre $R[c]$ du banc vérifie la propriété de linéarisabilité, définie dans la partie 2.1.3.
- *disponibilité partielle* : Pour chaque exécution possible, au moins 1 des χ registres du banc vérifie la propriété de disponibilité, c'est-à-dire que toute opération de Lecture ou d'Écriture sur ce registre par un processus correct termine.

Chacun des χ registres est indépendant des autres pour ce qui est des opérations, et donc des valeurs stockées et retournées par les lectures. Cependant, le partage de la disponibilité en présence d'asynchronisme impose d'utiliser chacun de ces registres

en parallèle si l'on souhaite une garantie de progression de l'algorithme. En effet, pour un processus qui a appelé une opération sur un de ces registres, il est impossible de savoir si cette opération va terminer ou non, quel que soit le temps déjà écoulé depuis l'appel à cause de l'asynchronisme. Un même processus peut donc appeler en parallèle une opération sur chacun des χ registres de ce banc, mais il lui est supposé impossible d'appeler plusieurs opérations sur un même registre coloré R_c simultanément : il doit attendre qu'une opération sur R_c termine avant d'en appeler une autre.

Cette idée d'utiliser plusieurs objets en un seul ensemble n'est pas nouvelle, déjà présente avec l'utilisation de vecteurs de registres partageant une opération de *snapshot* [3] qui permet de lire dans tous les registres à la fois. L'idée d'exécuter en parallèle plusieurs instances d'algorithmes n'est pas non plus inédite, présente par exemple dans [2]. La combinaison de ces deux principes amène naturellement à la création d'objets pouvant être utilisés en parallèle par plusieurs instances d'algorithmes, tout en vérifiant des propriétés communes, chacun des objet n'étant pas totalement indépendant.

Ainsi, même si chacun des registres d'un banc de χ registres colorés n'est pas vraiment utilisable seul, à cause de l'absence de la disponibilité, le banc dans son ensemble vérifie une propriété de disponibilité permettant son utilisation en pratique. De plus, on montre dans la partie 4.1.3 qu'il est possible d'implémenter un banc de χ registres si χ est suffisamment grand.

Si l'on souhaite utiliser plusieurs bancs de χ registres colorés à la fois (chaque banc ayant le même nombre χ de registres), il peut être souhaitable de disposer d'une propriété supplémentaire.

— *disponibilité partielle coordonnée* : Pour chaque exécution possible, il existe une couleur $c \in [1..\chi]$ telle que le c-ième registre (celui "de couleur c") de chaque banc vérifie la propriété de disponibilité.

Cette propriété reprend le principe de la disponibilité partielle en annonçant qu'au moins un registre (le c-ième) vérifie la propriété de disponibilité, mais cela est étendu en forçant les différents bancs à avoir une même *couleur* qui vérifie la disponibilité. Ainsi, avec une telle propriété, il n'est pas possible d'avoir une exécution dans laquelle seul le premier registre d'un banc vérifie la disponibilité, alors que seul le second d'un autre banc la vérifie.

4.1.2 Graphes de Kneser et quorums colorés

On considère maintenant un graphe de Kneser $KG_{n,k}$ [40, 64] qui est défini comme un graphe dont les sommets représentent les sous-ensembles de $[1..n]$ contenant exactement k éléments, et les arêtes relient deux sommets si et seulement si ces sommets correspondent à deux ensembles disjoints (c'est-à-dire ayant une intersection vide). Un tel graphe est représenté dans la figure 4.1.

La coloration propre d'un graphe [62] en utilisant χ couleurs revient à définir une fonction qui à chaque sommet du graphe associe une couleur dans $[1..\chi]$ de telle sorte que chaque arête du graphe relie deux sommets ayant des couleurs différentes. En particulier, lors d'une coloration du graphe de Kneser $KG_{n,k}$, chaque paire de sommet ayant une couleur identique est telle qu'aucune arête ne les relie, et donc les ensembles correspondant à ces sommets ont une intersection non-vide. Enfin, le nombre chromatique du graphe de Kneser $KG_{n,k}$ est $n - 2k + 2$ (si $k \leq n/2$), ce qui signifie qu'il est possible de trouver une χ-coloration de ce graphe pour $\chi = n - 2k + 2$ mais pas pour $\chi < n - 2k + 2$ [56].

Comme l'ensemble des processus est $\mathcal{P} = \{p_1, \ldots, p_n\}$, il est isomorphe à $[1..n]$, et donc chaque sommet d'un graphe de Kneser $KG_{n,k}$ peut représenter un ensemble de k

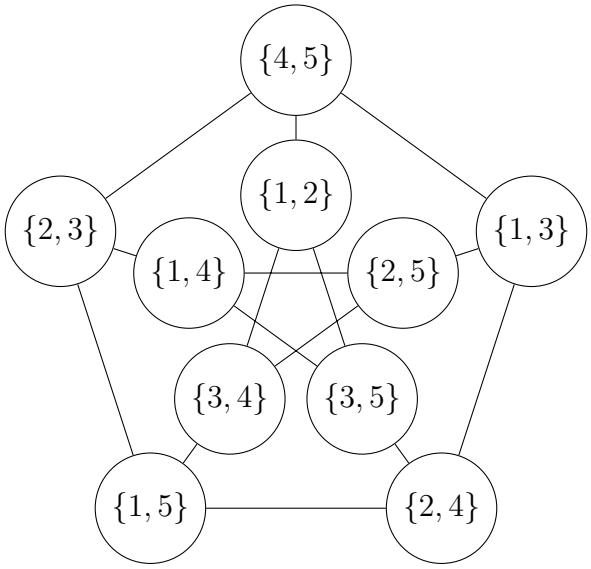

FIGURE 4.1 – Graphe de Kneser $KG_{5,2}$

processus du système. Ainsi, en choisissant $k = n - f$, le graphe correspondant $KG_{n,n-f}$ a pour sommet les *quorums* possibles de $n-f$ processus. Si on dispose d'une χ-coloration propre de ce graphe, on sait que pour toute paire de sommets qui sont de la même couleur, aucune arête ne relie ces 2 sommets. De plus, une arête relie 2 sommets de ce graphe si et seulement si les quorums que représentent ces sommets sont disjoints. Donc, 2 quorums-sommets partageant une couleur ont nécessairement une intersection non-vide. Ainsi, pour $\chi \geq n - 2(n - f) + 2 = 2f - n + 2$, on dispose alors d'une χ-coloration des quorums possibles de $n - f$ processus (c'est-à-dire une fonction qui à chaque quorum associe une couleur dans $[1..\chi]$) qui vérifie la propriété que toute paire de quorums qui partagent une couleur ont une intersection non-vide.

On appelle *fonction de χ-coloration de quorums* une fonction $\mathcal{C}$ de l'ensemble des quorums de $n - f$ processus (c'est-à-dire $\{Q \subseteq \mathcal{P} : |Q| = n - f\}$) dans $[1..\chi]$, qui vérifie : $\forall Q, Q' \subseteq \mathcal{P}, \mathcal{C}(Q) = \mathcal{C}(Q') \Rightarrow Q \cap Q' \neq \emptyset$. Une telle coloration de quorums est par exemple illustrée dans la figure 4.2.

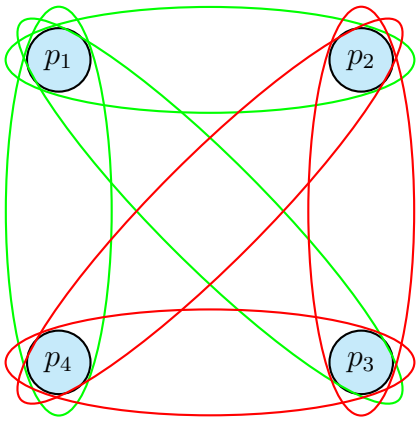

FIGURE 4.2 – Coloration des quorums pour $n = 4$ et $f = n - f = 2$

Comme expliqué dans le paragraphe précédent, on sait qu'il existe une telle fonction pour $\chi \geq 2f - n + 2$ lorsque $f \geq n/2$. De plus, dans les mêmes conditions, il n'existe pas de telle fonction avec $\chi < 2f - n + 2$. En effet, si une telle fonction existait, alors il serait possible de colorer chaque sommet du graphe de Kneser $KG_{n,n-f}$ avec χ couleurs, de telle sorte que 2 quorums-sommets de même couleur aient une intersection non-vide,

c'est-à-dire (par définition du graphe de Kneser) qu'aucune arête ne relie ces 2 sommets. Ainsi, cela formerait une χ-coloration propre du graphe $KG_{n,n-f}$ avec $n - f \leq n/2$ et $\chi < n-2(n-f)+2$, ce qui n'est pas possible d'après la définition du nombre chromatique de ce graphe, qui est le nombre *minimal* de couleurs nécessaire pour réaliser une coloration propre du graphe.

Pour revenir au banc de χ registres *colorés*, on montrera dans la partie suivante qu'il est possible d'en implémenter si les processus disposent d'une fonction commune de χ-coloration de quorums $\mathcal{C}$. On associera alors à chacun des χ registres une *couleur* unique parmi $[1..\chi]$, qui aura alors un lien avec les couleurs des quorums.

Cette idée de colorer des quorums ressemble au *détecteur de défaillances* $V\Sigma_k$ présenté dans l'article [33]. La propriété d'intersection de quorums de même couleur est très similaire, et celle qui garantit qu'au bout d'un certain temps les quorums ne contiennent que des processus corrects est implicitement vraie, de par le fait que seuls les processus qui ne sont pas tombés en panne peuvent participer aux communications (et donc faire partie des quorums), et que les processus qui ne tombent jamais en panne sont corrects (par définition).

4.1.3 Implémentation

Il est possible de simuler un ensemble de registres χ-colorés si $\chi \geq 2f - n + 2 = n - 2 * (n - f - 1)$, et si les processus disposent d'une même fonction de coloration de quorums $\mathcal{C}$. Une telle fonction peut être qualifiée d'*oracle*, car elle permet aux différents processus d'avoir des informations en commun sans communication, et parce qu'on ne s'intéresse pas à son implémentation (on la considère comme une "boîte noire"). À titre d'exemple, si les processus connaissent chacun les identifiants id_j de tous les processus, alors il est aisé de construire une telle fonction de manière déterministe avec $\chi = 2f-n+2$ (et chaque processus aura ainsi accès à la même fonction).

Lemme 4.1.1. *Si chaque processus du système connaît l'ensemble* $Id = \{id(p_i), \forall p_i \in \mathcal{P}\}$ *des identifiants des processus, alors chaque processus peut simuler une même fonction de* χ*-coloration de quorums avec* $\chi = 2f - n + 2$.

Démonstration. Une possibilité est d'ordonner les processus par identifiant, puis de les associer chacun à un entier dans $[1..n]$ (par ordre croissant d'identifiant), appelé *rang*. Ensuite on considère qu'un quorum dont le plus *petit* processus (en utilisant cet ordre) a pour rang i aura pour couleur i si $i < 2f - n + 2$, et aura pour couleur $2f - n + 2$ si $i \geq 2f - n + 2$. Ainsi, chaque quorum de couleur $i < 2f - n + 2$ contient le processus de rang i et a donc une intersection non-vide avec chaque autre quorum de même couleur. Et chaque quorum de couleur $2f - n + 2$ contient uniquement des processus de rang dans $[2f - n + 2..n]$, qui est un ensemble de $2(n-f)-1$ processus, donc toute paire de quorums dans ce sous-ensemble a une intersection non-vide, car $n - f < (2(n - f) - 1)/2$. □

Chacun des χ registres est associé à une *couleur* unique $c \in [1..\chi]$, et on notera alors R_c ce registre. L'algorithme 4.1, inspiré de la simulation ABD de registres atomiques (présentée en partie 3.1.2), implémente un banc de registres χ-colorés, où on suppose que les processus connaissent le nombre χ de couleurs, et disposent tous du même oracle $\mathcal{C}$ associant chaque quorum à une couleur dans $[1..\chi]$. Lors de communication pour une opération sur le registre R_c, au lieu d'attendre simplement d'obtenir un ensemble de réponses d'au moins $n - f$ processus (quorum) comme dans ABD, on attend d'obtenir un quorum *de couleur c*. Ainsi, si les $n - f$ premières réponses forment un quorum d'une

autre couleur $c' \neq c$, le processus attend d'obtenir d'autres réponses, jusqu'à ce qu'un sous-ensemble de $n - f$ processus parmi tous les processus ayant répondu forme un quorum de couleur c.

Algorithme 4.1 Implémentation d'un banc de χ registres colorés (code exécuté par p_i)

1: **Initialisation**
2: **pour** c **de** 1 **à** χ **faire**
3: $\langle ts_i[c], v_i[c] \rangle \leftarrow \langle 0, \perp \rangle$;
4: $seq_i \leftarrow 0$;
5: **Fonction** LECTURE(c) ▷ correspond au registre R_c
6: COMMUNIQUE(L_1, c, $\langle ts_i[c], v_i[c] \rangle$) ;
7: $\langle ts, v \rangle \leftarrow \langle ts_i[c], v_i[c] \rangle$;
8: COMMUNIQUE(L_2, c, $\langle ts, v \rangle$) ;
9: **retourner** v ;
10: **Procédure** ECRITURE(c, v) ▷ correspond au registre R_c
11: $\langle ts_e[c], v_e[c] \rangle \leftarrow \langle ts_e[c] + 1, v \rangle$;
12: COMMUNIQUE(E, c, $\langle ts_e, v_e \rangle$) ;
13: **Quand** un message m **est reçu** du processus p_j
14: **si** m est de type (L_1, c, seq) **alors**
15: **envoyer** (Re, c, seq, $\langle ts_i[c], v_i[c] \rangle$) **à** p_j ;
16: **sinon si** m est de type (L_2, c, seq, $\langle ts, v \rangle$) ou (E, c, seq, $\langle ts, v \rangle$) **alors**
17: **si** $ts > ts_i[c]$ **alors**
18: $\langle ts_i[c], v_i[c] \rangle \leftarrow \langle ts, v \rangle$;
19: **envoyer** (Re, c, seq) **à** p_j ;
20: **Procédure** COMMUNIQUE($Type$, c, $\langle ts_0, v_0 \rangle$)
21: $seq_i \leftarrow seq_i + 1$;
22: $Reponses \leftarrow \emptyset$;
23: $Quorum \leftarrow \emptyset$;
24: **si** $Type = L_1$ **alors**
25: **pour chaque** processus p du système **faire**
26: **envoyer** ($Type$, c, seq_i) **à** p ;
27: **tant que** $Quorum = \emptyset$ **faire**
28: **attendre** de recevoir un message (Re, c, seq_i, $\langle ts, v \rangle$) d'un processus p_j ;
29: $Reponses \leftarrow Reponses \cup \{p_j\}$;
30: **si** $ts > ts_i[c]$ **alors**
31: $\langle ts_i[c], v_i[c] \rangle \leftarrow \langle ts, v \rangle$;
32: $Quorum =$ SOUS-ENSEMBLE($Reponses$, c) ;
33: **sinon si** $Type = L_2$ ou $Type = E$ **alors**
34: **pour chaque** processus p du système **faire**
35: **envoyer** ($Type$, c, seq_i, $\langle ts_0, v_0 \rangle$) **à** p ;
36: **tant que** $Quorum = \emptyset$ **faire**
37: **attendre** de recevoir un message (Re, c, seq_i) d'un processus p_j ;
38: $Reponses \leftarrow Reponses \cup \{p_j\}$;
39: $Quorum =$ SOUS-ENSEMBLE($Reponses$, c) ;
40: **Fonction** SOUS-ENSEMBLE(Ens, c)
41: **pour chaque** sous-ensemble $Q \subseteq Ens$ tel que $|Q| = n - f$ **faire**
42: **si** $\mathcal{C}(Q) = c$ **alors** ▷ Où $\mathcal{C}$ est l'oracle associant chaque quorum à une couleur
43: **retourner** Q ;
44: **retourner** $\emptyset$;

On démontre désormais que cet algorithme implémente bien un banc de registres χ-colorés, et on peut noter que certaines des preuves des lemmes suivants sont inspirées de celles de lemmes présentés en partie 3.1.2.

Lemme 4.1.2. *Pour toute exécution, il existe une couleur $c_0 \in [1..\chi]$ telle que toute appel à la fonction Communique(_, c_0, _) par un processus correct termine.*

Démonstration. Pour toute exécution, au plus f processus sont susceptibles de tomber en panne, donc au moins $n - f$ processus sont corrects. Considérons un ensemble Q_0 de $n - f$ processus corrects pour une exécution donnée, et notons c_0 la couleur de ce quorum (c'est-à-dire $\mathcal{C}(Q_0)$). Soit p un processus correct qui appele la fonction Communique(_, c_0, _) au cours de cette exécution, et supposons que cet appel ne termine pas. Alors, cela signifie que la boucle *tant que* de cette fonction ne termine pas (ligne 27 ou 36 selon le $Type$ passé en paramètre). Ainsi, si le processus p passe un temps infini dans cette boucle, il finit par recevoir toutes les réponses qui lui ont été envoyées pour son message de début de fonction (ligne 26 ou 35). En particulier, p finit par recevoir une réponse de chaque processus correct, dont chaque processus de Q_0. On a alors, au bout d'un certain temps, $Q_0 \subseteq Reponses$, et $|Q_0| = n - f$. Donc, lors de l'appel à la fonction Sous-Ensemble(), la variable Q prend pour valeur Q_0 (ligne 41) dans la boucle *pour* sauf si cette boucle termine afin d'avoir énumérer tous les sous-ensembles de Ens de taille $n - f$. Dans le cas où Q_0 apparait, comme $\mathcal{C}(Q_0) = c_0$, la fonction retourne $Q_0 \neq \emptyset$. Dans le cas où la boucle *pour* termine plus tôt, c'est qu'un autre sous-ensemble Q a vérifié $\mathcal{C}(Q) = c_0$, et que la fonction a retourné $Q \neq \emptyset$. Dans chacun de ces deux cas, on observe alors, dans la fonction communique, une variable $Quorum \neq \emptyset$. Donc la boucle *tant que* termine, ce qui contredit notre hypothèse et prouve ce lemme. □

Corollaire 4.1.3. *Pour toute exécution, il existe une couleur $c_0 \in [1..\chi]$ telle que tout appel à une opération de Lecture ou d'Ecriture sur R_{c_0} par un processus correct termine. C'est-à-dire que le banc de χ registres colorés vérifie la propriété de* disponibilité partielle.

Lemme 4.1.4. *Lors de n'importe quelle exécution, les valeurs successives de la variable locale $ts_i[c]$ du processus p_i sont croissantes, et ce pour chaque couleur c.*

Démonstration. Les seuls moments où cette variable est modifiée sont aux lignes 11, 18, et 31, qui vérifient bien que la nouvelle valeur est plus grande que la précédente. □

Lemme 4.1.5. *Lorsqu'une exécution C de la fonction Communique(_, c, $\langle ts_0, v_0 \rangle$) de type L_2 ou E termine, un quorum Q de $n - f$ processus tel que $\mathcal{C}(Q) = c$ vérifie, pour chacun de ses processus, $ts_i[c] \geq ts_0$, où les $ts_i[c]$ sont les variables locales de ces processus.*

Démonstration. A la fin de C, la variable $Quorum$ contient $n - f$ processus tels que chacun de ces processus a reçu le message envoyé au début de C (ligne 35) et y a répondu par un message de type Re. De plus, lors de la réception de ce message contenant $\langle ts_0, v_0 \rangle$ et c, un processus p_i stocke ts_0 dans sa variable $ts_i[c]$ si $ts_0 > ts_i[c]$ (ligne 18). Enfin, les valeurs successives des variables $ts_i[c]$ sont croissantes. Donc, chaque processus p_i qui reçoit le message vérifie $ts_i[c] \geq ts_0$. Cela signifie que tous les processus de l'ensemble $Quorum$ vérifient cette propriété. Finalement, la variable $Quorum$ contient un quorum de $n - f$ processus vérifiant $\mathcal{C}(Quorum) = c$, ce qui prouve ce lemme. □

Lemme 4.1.6. *Lorsqu'une exécution C de la fonction Communique(L_1, c, _) par un processus p_j termine, un quorum Q de $n - f$ processus tel que $\mathcal{C}(Q) = c$ vérifie, pour chaque de ses processus, $ts_j[c] \geq ts_i^0[c]$, où les $ts_i^0[c]$ sont les valeurs des variables locales $ts_i[c]$ de ces processus lors de l'appel à cette fonction Communique().*

Démonstration. A la fin de C, la variable *Quorum* contient $n - f$ processus tels que chacun de ces processus a reçu le message envoyé au début de C (ligne 26) et y a répondu par un message de type *Re*. Chacune de ces réponses contient une valeur $ts_i^1[c]$, qui est la valeur de $ts_i[c]$ au moment où p_i a envoyé cette réponse (ligne 15). Comme les valeurs stockées dans les variables $ts_i[c]$ sont croissantes, $ts_i^0[c] \leq ts_i^1[c]$. Lorsque le processus p_j reçoit une telle réponse, si $ts_j[c] < ts_i^1[c]$ alors $ts_j[c]$ prend pour nouvelle valeur $ts_i^1[c]$ (ligne 31). En particulier, à la fin de C, on a $ts_j[c] \geq ts_i^1[c]$ pour chaque $ts_i^1[c]$ reçu par un message d'un des processus de *Quorum*. Donc, pour chaque processus p_i dans *Quorum*, on a $ts_j[c] \geq ts_i^0[c]$, et $\mathcal{C}(Quorum) = c$, ce qui prouve le lemme. □

Corollaire 4.1.7. *Supposons qu'une exécution C de la fonction Communique(_, c, $\langle ts_0, v_0 \rangle$) de type L_2 ou E termine avant qu'une autre exécution C' de Communique(_, c, _) de type L_1 ne soit appelée par un processus p_j. Alors, à la fin de l'exécution C' si celle-ci termine, on a $ts_j[c] \geq ts_0$, où $ts_j[c]$ est la variable locale de p_j.*

Démonstration. On note Q l'ensemble *Quorum* de C et Q' celui de C'. D'après les deux lemmes précédents, on a, à la fin de C, $\forall p_i \in Q$, $ts_i[c] \geq ts_0$, et à la fin de C', $\forall p_i \in Q'$, $ts_j[c] \geq ts_i^0[c]$. Or, comme C termine avant que C' ne commence, on a $\forall p_i \in Q \cap Q'$, $ts_0 \leq ts_i^0[c]$. Donc, si $Q \cap Q' \neq \emptyset$, à la fin de C', $ts_j[c] \geq ts_0$ par transitivité. Or, on sait que $\mathcal{C}(Q) = \mathcal{C}(Q') = c$, et donc, par définition de $\mathcal{C}$, $Q \cap Q' \neq \emptyset$, ce qui prouve ce corollaire. □

Corollaire 4.1.8. *Chaque registre R_c définit par l'algorithme ci-dessus vérifie la propriété de linéarisabilité.*

Démonstration. La preuve de cette propriété est identique à celle des lemmes 3.1.7 et 3.1.8, présentée dans le chapitre 3 pour la simulation ABD, grâce aux lemmes et corollaires précédents qui sont similaires à ceux utilisés dans la démonstration en question. □

Théorème 4.1.9. *L'algorithme 4.1 implémente un banc de registres χ-colorés.*

Lors de l'implémentation de *plusieurs* bancs de χ registres colorés, à l'aide de l'algorithme précédent copié pour chaque registre de chaque banc, on vérifie en plus la propriété de disponibilité partielle coordonnée. En effet, le lemme 4.1.2 indique qu'il existe une couleur c_0 telle que chaque appel à la fonction Communique avec cette couleur termine, en se basant simplement sur l'existence d'un quorum arbitraire Q_0 de processus corrects. Or, chaque banc est implémenté en utilisant une telle fonction Communique, et le même quorum Q_0 peut être choisi pour prouver le lemme sur chacune des fonctions Communique des différents bancs. Ainsi, il existe une même couleur c_0 telle que la fonction Communique de chaque banc vérifie le lemme 4.1.2. Cela suffit bien à démontrer qu'il existe une même couleur c_0 telle que le registre R_{c_0} de chaque banc vérifie la disponibilité, d'où :

Propriété 4.1.10. *Si cet algorithme est utilisé pour implémenter plusieurs bancs de χ registres colorés (avec un même χ issu de la fonction de coloration $\mathcal{C}$), alors ces bancs vérifient la propriété de* disponibilité partielle coordonnée.

4.1.4 Limites et utilisation

Il a été démontré qu'un banc de χ registres colorés pouvait être implémenté si $\chi \geq 2f - n + 2$ et que les processus partagent une fonction $\mathcal{C}$ de χ-coloration de quorums, et aussi plus généralement si les processus connaissent les identifiants de chaque processus. De plus, une fonction de χ'-coloration de quorums ne peut pas exister pour $\chi' < 2f - n + 2$. Bien que cela démontre qu'une telle méthode ne peut pas être utilisée pour simuler des

bancs de χ' registres colorés, cela ne suffit malheureusement pas à prouver que de tels objets ne peuvent être implémentés.

On souhaite maintenant utiliser de tels bancs de registres dans un algorithme distribué. La première idée d'utilisation est de partir d'un algorithme distribué utilisant des registres atomiques classiques, et de remplacer ces registres par des bancs de χ registres colorés. Ainsi, si chacun des n algorithmes locaux est un algorithme séquentiel, on remplace chaque opération de Lecture (ou d'Ecriture) sur un registre R par l'exécution en parallèle de χ Lectures (ou Ecritures) sur chacun des χ registres du banc associé, et dès qu'une de ces exécution termine (ce qui arrive nécessairement si le processus est correct), on en récupère le résultat. Ainsi, on remplace un exemple d'algorithme du premier type par le second :

Fonction ALGORITHME CLASSIQUE
...
$a \leftarrow$ LECTURE( ) sur le registre R_1 ;
ECRITURE(a) sur le registre R_2 ;
...
Fonction ALGORITHME MODIFIÉ
...
pour chaque $c \in [1, \chi]$ **faire**
Exécute en parallèle LECTURE(c) sur le banc B_1
attendre qu'une de ces opérations Lecture(c_1) termine en retournant a_1 ;
$a \leftarrow a_1$;
les autres opérations de Lecture en cours sont interrompues ou simplement ignorées
pour chaque $c \in [1, \chi]$ **faire**
Exécute en parallèle ECRITURE(c, a) sur le banc B_2
attendre qu'une de ces opérations Ecriture(c_2, a) termine ;
...

Cependant, une telle modification risque de ne pas atteindre le but souhaité. En effet, il est possible que pour chaque appel à une opération par un processus p_i, des registres de différentes couleurs (y compris pour un même banc) terminent en premier. Ainsi, les résultats peuvent être incohérents par rapport au comportement attendu. Mais le meilleur moyen d'illustrer cela est par un exemple simple.

On considère le problème (fictif) suivant : Le processus p_n obtient en entrée une suite infinie de nombres (x_1, x_2, ...). Chaque autre processus peut retourner une valeur *vrai* si il sait que $\exists i, j \in \mathbb{N}^* : x_i > x_{i+j}$, c'est-à-dire que la suite de nombres possédée par p_n n'est pas croissante. Mais un tel processus ne peut pas terminer si la suite des x_i est croissante. Un algorithme (n'ayant qu'un intérêt pédagogique) permettant de résoudre ce problème en utilisant un registre atomique 1WnR (avec p_n pour écrivain) est l'algorithme 4.2 :

Algorithme 4.2 Exemple d'algorithme à registre atomique illustrant un problème possible du passage aux registres χ-colorés

```
1: Le registre R est initialisé à −∞.
2: Fonction POUR p_n((x_1, ...))
3:     i ← 0 ;
4:     tant que vrai faire
5:         ECRITURE(x_i) sur le registre R ;
6:         i ← i + 1 ;
7: Fonction POUR p_i
8:     a, b ← −∞ ;
9:     tant que a ≤ b faire
10:        a ← b ;
```

11: $b \leftarrow$ Lecture( ) sur le registre R ;

12: **retourner** $vrai$;

Avec un registre atomique classique, cet algorithme fonctionne, car si un processus p_i appelle Lecture() deux fois de suite, la seconde retourne nécessairement une valeur au moins aussi récente que la première. Comme les valeurs écrites dans le registre sont les x_i écrits dans l'ordre, cela signifie que si $a > b$, alors les deux Lecture() correspondantes ont retourné x_j et x_{j+k} avec $x_j > x_{j+k}$.

Considérons maintenant cet algorithme dans lequel on utilise un banc B de χ registres colorés à la place du registre R (tout en considérant que chacun des registres du banc sont initialisés à $-\infty$), et que chaque opération est remplacée par l'exécution en parallèle de χ opérations s'arrêtant dès qu'une de ces opérations termine. On considère alors une exécution particulière, pour laquelle la suite des x_i est strictement croissante (donc aucun processus ne devrait retourner $vrai$) :

Le processus p_n commence par exécuter Ecriture(c, x_1) pour tout c, et l'opération pour $c = 1$ termine en premier, tandis que celle pour $c = 2$ termine peu après ou est interrompue. Ensuite, p_n exécute Ecriture(c, x_2) pour tout c, et l'opération pour $c = 2$ termine. Le processus p_1 appelle alors Lecture(c) pour tout c, et l'opération pour $c = 2$ termine, retournant donc la valeur x_2, stockée dans b. L'opération pour $c = 1$ termine ou est interrompue, mais dans tous les cas sa valeur est ignorée. Ensuite, p_1 stocke x_2 dans a, puis appelle à nouveau Lecture(c) pour tout c, et l'opération pour $c = 1$ termine, retournant x_1. En effet, cela est possible car $B[1]$ vérifie la linéarisabilité, et que l'opération Ecriture(1, x_1) est terminée (donc la valeur retournée est au moins aussi récente que x_1), tandis que l'opération Ecriture(1, x_2) n'a pas encore terminé. Donc, p_1 stocke la valeur x_1 dans b. Ensuite, p_1 teste si $a \leq b$, et remarque que $a = x_2 > x_1 = b$, donc retourne $vrai$. Cela n'aurait pas dû se produire, car la suite des x_i est strictement croissante, donc cet algorithme ne fonctionne pas comme attendu.

Une solution générale pour ce problème est d'avoir une approche légèrement différente. Au lieu de garder l'algorithme local de chaque processus intact, avec pour seule différence les opérations sur les registres, qui sont dupliquées en χ copies parallèles, il est possible de dupliquer l'algorithme dans son ensemble en χ copies parallèles (y compris les variables locales, notamment), en vérifiant que chaque copie de l'algorithme local n'utilise qu'une seule couleur de registre. Ainsi, on a une transformation de l'algorithme du type suivant :

Fonction Algorithme classique

...

$a \leftarrow$ Lecture( ) sur le registre R_1 ;

Ecriture(a) sur le registre R_2 ;

...

Fonction Algorithme modifié

pour chaque $c \in [1, \chi]$ **faire**

Exécuter en parallèle l'algorithme local coloré suivant, de couleur c ;

Attendre que qu'un de ces χ algorithmes locaux termine.

Si il retourne une valeur v, alors retourner cette valeur.

Fonction Algorithme local coloré(c)

...

$a \leftarrow$ Lecture(c) sur le banc B_1 ;

Ecriture(c,a) sur le banc B_2 ;

...

La figure 4.3 permet de visualiser les deux façons ainsi présentées de modifier un algorithme pour utiliser des bancs de registres χ-colorés.

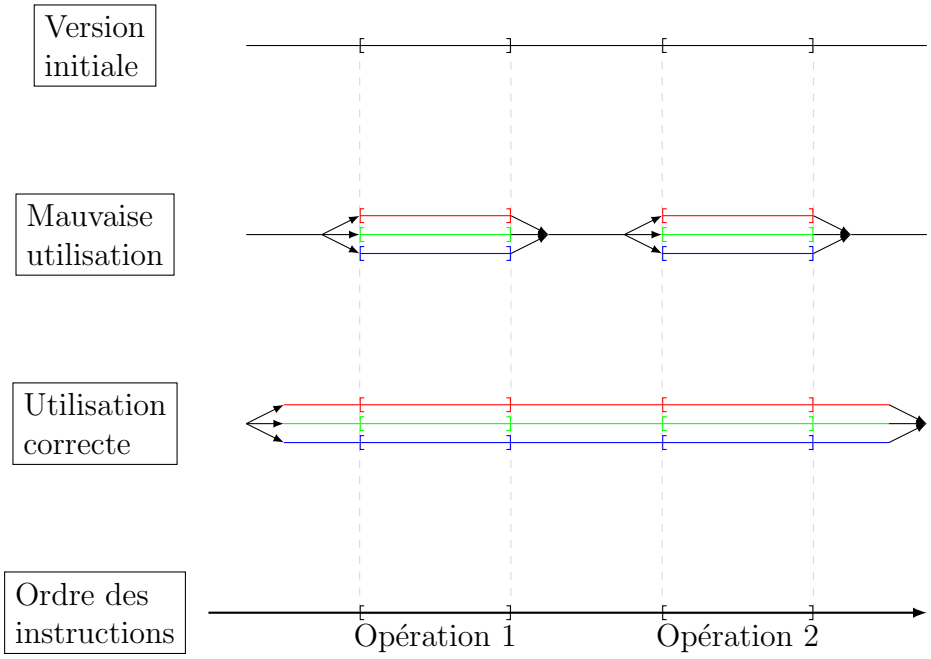

FIGURE 4.3 – Utilisation des bancs de registres χ-colorés ($\chi = 3$)

De cette manière, chaque version de l'algorithme local a une couleur c, et reste cohérente, car n'utilise que des registres d'une même couleur c qui vérifient tous la propriété de linéarisabilité. Le seul défaut de ces registres est qu'ils peuvent ne pas vérifier la propriété de disponibilité, ce qui signifie qu'un tel algorithme local coloré peut ne pas terminer. Mais comme, pour au moins une couleur c_0, chaque registre $B[c_0]$ vérifie la disponibilité (grâce à la disponibilité partielle coordonnée), cela signifie que si l'algorithme classique (utilisant des registres atomiques) terminait nécessairement, alors l'algorithme local de couleur c_0 termine aussi.

Dans un tel contexte de duplication d'algorithmes, il est possible que pour un processus p_i, l'algorithme local de couleur c_i termine en premier, alors que pour p_j il s'agit de l'algorithme c_j, avec $c_i \neq c_j$. Dans ces conditions, les valeurs de retour de p_i et p_j risquent de ne pas être (ensemble) conformes avec les spécifications du problème pour lequel l'algorithme classique a été réalisé. Cependant, pour chaque couleur c, l'ensemble des valeurs de retour des processus dont l'algorithme c a terminé en premier est correct vis-à-vis du problème en question, car pour cette version c de l'algorithme, tout se passait comme l'algorithme classique.

Par exemple, si l'on suppose qu'il existe un algorithme permettant de résoudre le consensus à l'aide de registres atomiques, et ce malgré les pannes et l'asynchronisme, alors on pourrait réaliser simplement un algorithme de χ-accord ensembliste. En effet, pour chaque couleur $c \in [1, \chi]$, les propriétés de l'algorithme classique seraient conservées dans l'algorithme c, et donc les valeurs de retour v des processus terminant avec l'algorithme c seraient identiques.

4.2 Affaiblir la cohérence : les α-registres

4.2.1 Motivation et inspirations

L'idée d'affaiblir les propriétés de cohérence afin de contourner la barrière d'une majorité de pannes (et du théorème CAP) n'est pas nouvelle, comme cela a été expliqué dans le chapitre 3. Afin de bien caractériser un objet partagé similaire au registre atomique mais pouvant être implémenté en présence d'une majorité de pannes, il convient d'analyser

les différentes propriétés souhaitables, pas à pas. Pour commencer, on souhaite conserver la propriété de terminaison des différentes opérations, ce qui peut être garanti par un algorithme qui limite ses communications à des quorums de $n - f$ processus (c'est-à-dire qui fonctionne en envoyant chaque message à tous les processus, et en attendant jusqu'à $n - f$ réponses), à la façon de la simulation ABD.

Ensuite, la propriété de cohérence finale peut être garantie assez simplement pour tout registre mono-écrivain. En effet, il est alors possible de totalement ordonner les valeurs écrites, ce qui élimine la possibilité de conflits pour décider quelle valeur est plus récente qu'une autre. Cependant, la propriété de cohérence finale n'apporte que peu de cohérence au système (ne garantissant qu'une stabilisation du système au bout d'un certain temps s'il n'y a qu'un nombre fini de lectures), et il est souhaitable d'y ajouter quelques propriétés.

Des propriétés locales, inspirées de [92], peuvent être maintenues simplement, à l'aide de l'utilisation d'estampilles croissantes. Ainsi, on peut garantir la *monotonie des lectures*, la *non-trivialité des lectures*, ainsi que la *visibilité locale des écritures*, qui sont des propriétés de cohérence locales simples et intuitives. Ces propriétés, définies dans la sous-partie suivante, sont notamment vérifiées par l'algorithme de la simulation ABD s'il était directement exporté dans le modèle avec une majorité de pannes. Enfin, ces propriétés sur un tel registre mono-écrivain seul permettent de vérifier directement la propriété de cohérence causale.

Afin de pallier à la faiblesse de la propriété de cohérence finale en cas d'une infinité d'écritures, nous avons ajouté une propriété de *propagation*, définie dans la sous-partie suivante. Elle permet de garantir une progression des données lues lors d'écritures illimitées, et englobe aussi la propriété de cohérence finale dans le cas d'un nombre fini d'écriture, à condition d'avoir un registre mono-écrivain. Autrement dit, cette propriété garantit que, en cas d'écritures infinies, même si les valeurs lues peuvent être arbitrairement *vieilles* (en terme de nombre d'écritures entre celle-ci et la dernière effectuée), chaque processus correct finit par lire des valeurs de plus en plus récentes.

Une propriété supplémentaire que l'on pourrait souhaiter est celle de *k-obsolescence bornée*, qui a été évoquée et définie dans la partie 3.2.1. Cependant, il est impossible de garantir cette propriété dans un modèle avec une majorité de pannes et pour lequel les propriétés précédentes sont vérifiées.

En effet, si une première valeur v_1 est écrite, puis qu'aucune écriture n'est effectuée pendant un temps arbitrairement grand, chaque processus finira par lire v_1 lors de ses lectures au bout d'un certain temps t_0, de par la cohérence finale. Ensuite, si N écritures sont effectuées après t_0 par le processus écrivain de telle sorte que seule une moitié des processus (au plus) obtienne les données correspondantes, les messages vers l'autre moitié étant ralentis arbitrairement, ces écritures doivent terminer en temps fini par propriété de terminaison. Enfin, chaque processus de l'autre moitié qui décide d'effectuer une lecture doit voir cette opération retourner, et à cause de la monotonie des lectures et du délai de communication arbitraire, il doit retourner v_1. Ainsi, il est impossible d'éviter qu'une lecture puisse retourner une valeur plus ancienne que les N dernières valeurs écrites, et ce pour tout N. Donc, pour tout entier k, il est impossible de garantir la k-obsolescence bornée.

Cependant, une propriété assez proche mais plus faible *peut* être vérifiée dans ce contexte : la *lecture α-bornée*. En effet, en ne bornant plus l'*âge* (en terme de nombre d'écritures) des valeurs lues, mais le nombre de valeurs différentes lues, on obtient une propriété non-triviale mais pouvant être garantie. Il peut être intéressant de noter que la simulation ABD exportée dans un contexte avec une majorité de pannes ne vérifie pas cette propriété.

4.2.2 Définition

Un α-registre est un objet partagé, disposant des opérations Lecture() et Ecriture(), qui nécessite d'être utilisé en vérifiant :

1. Un processus ne peut appeler une opération Lecture() ou Ecriture() que si sa dernière invocation à une de ces opérations (si elle existe) a terminé.

Et cet objet est défini par les propriétés suivantes :

1. *Terminaison* : Chaque appel à une opération Lecture() ou Ecriture() par un processus correct termine.
2. *Non-trivialité* : Chaque opération de Lecture() qui termine, retourne $\perp$ ou une valeur v telle qu'une opération Ecriture(v) a été appelée avant la terminaison de cette Lecture.
3. *Monotonie des Lectures* : Soient L, L' deux opérations de Lecture() exécutées par un même processus p, telles que L précède L', et que L retourne v et L' retourne v'. Si $v \neq \perp$, alors $v' \neq \perp$, et l'opération Ecriture(v) ayant écrit v précède l'opération Ecriture(v') ayant écrit v', ou ces deux opérations sont concurrentes ou égales.
4. *Visibilité des Ecritures* : Soit L une opération de Lecture exécutée par un processus p et retournant v. Si p a exécuté une opération d'Ecriture() qui précède L, alors $v \neq \perp$. De plus, si E est la dernière Ecriture précédant L exécutée par p, alors E précède l'opération Ecriture(v) ayant écrit v, ou ces deux opérations sont concurrentes ou égales.
5. *Cohérence finale* : Si lors d'une exécution, à partir d'un temps t_0, aucune opération d'Ecriture n'est appelée, que toutes les opérations d'Ecriture ont terminé, et qu'une infinité d'opérations de Lecture sont exécutées, alors au bout d'un certain temps t_1, toute opération de Lecture qui termine retourne une même valeur v.
6. *Propagation* : Soit $v \neq \perp$ la valeur de retour d'une opération de Lecture() *op* ou le paramètre d'une opération d'Ecriture() *op*, exécutée par un processus p correct. Si une infinité d'opérations de Lecture() sont exécutées, alors, à partir d'un certain temps t_0, toute opération de Lecture() qui termine retourne une valeur v' (possiblement différente à chaque fois) vérifiant $v' \neq \perp$ et l'opération Ecriture(v) précède l'opération Ecriture(v'), ou ces deux opérations sont concurrentes ou égales.
7. *Lecture α-bornée* : Soit un intervalle de temps (possiblement infini) $\mathcal{I}$ au cours d'une exécution quelconque. Soit $\mathcal{L}$ l'ensemble des Lecture() appelées après le début de $\mathcal{I}$ et terminées avant la fin de $\mathcal{I}$, et V_L l'ensemble des valeurs retournées par ces Lecture(). Soit $\mathcal{E}$ l'ensemble des Ecritures appelées avant la fin de $\mathcal{I}$ et ne terminant pas avant le début de $\mathcal{I}$, et V_E l'ensemble des valeurs paramètre de ces fonctions. Alors, $V_v = V_L \setminus V_E$, l'ensemble des *vieilles* valeurs lues pendant $\mathcal{I}$, vérifie $|V_v| \leq \alpha$.
8. *Mono-écrivain* : Un seul processus p_e appelé processus *écrivain* peut appeler l'opération d'écriture.

Un corollaire de la propriété de lecture α-bornée est que, lors d'une exécution où aucune Ecriture n'a lieu après un temps t_0, toute Lecture appelée après t_0 retourne une valeur parmi au plus α différentes possibles, car alors $V_E = \emptyset$ et $V_L = V_v$.

Enfin, toutes les propriétés de l'α-registre, sauf la dernière, sont formulées comme si plusieurs processus pouvaient appeler l'opération d'Ecriture(). Malheureusement, il est impossible en présence d'une majorité de pannes d'implémenter un registre multi-écrivains vérifiant toutes ces propriétés, comme cela sera expliqué dans la partie 4.5.

4.3 Simuler des α-registres

Il est possible d'implémenter des α-registres avec $\alpha = 2(2f - n + 2) - 1$. Une première version, présentée dans [29], est un algorithme dit *bruyant*, car des messages sont constamment échangés entre les processus même lorsqu'aucune opération n'est en cours. Il est possible de modifier cette version pour établir une seconde version dite *silencieuse*, car seul un nombre borné de messages peut être envoyé en absence d'opération en cours.

4.3.1 Version bruyante

Algorithme 4.3 Algorithme bruyant d'implémentation d'α-registre (code exécuté par p_i)

1: **Initialisation**
2: $seq_i \leftarrow 1$; $\langle v_i, ts_i \rangle \leftarrow \langle \perp, 0 \rangle$; $\langle vl_i, tsl_i \rangle \leftarrow \langle \perp, 0 \rangle$;
3: $Ql_i \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$;
4: $Accept_i[1..n] \leftarrow [2, \ldots, 2]$; ▷ tableau de n entiers initialisés à 2
5: **pour chaque** processus p_j **faire**
6: **envoyer** $(seq_i, \langle v_i, ts_i \rangle, 0)$ **à** p_j ;
7: **Procédure** ECRITURE(v)
8: $\langle v_i, ts_i \rangle \leftarrow \langle v, ts_i + 1 \rangle$; $\langle vl_i, tsl_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $seq_i \leftarrow seq_i + 1$; $Q_i \leftarrow \emptyset$;
9: **attendre** jusqu'à ce que $|Q_i| \geq n - f$;
10: **Fonction** LECTURE( )
11: $n\_iter \leftarrow 0$;
12: **répéter**
13: $\langle vl_i, tsl_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $seq_i \leftarrow seq_i + 1$; $Ql_i \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$;
14: **attendre** jusqu'à ce que $|Ql_i \cup Q_i| \geq n - f$;
15: $n\_iter \leftarrow n\_iter + 1$;
16: **jusqu'à ce que** $(|Q_i| \geq n - f)$ ou $(n\_iter \geq N)$ ▷ $N = (2f + 1)(2f + 3) + 1$
17: **retourner** vl_i ;
18: **Quand** le message $(seq, \langle v, ts \rangle, ancien\_seq)$ **est reçu** de p_j
19: **si** $ancien\_seq = seq_i$ **alors**
20: **si** $ts > tsl_i$ **alors** $Ql_i \leftarrow Ql_i \cup \{p_j\}$;
21: **si** $ts = tsl_i$ **alors** $Q_i \leftarrow Q_i \cup \{p_j\}$;
22: **si** $ts > ts_i$ **alors**
23: **si** $Accept_i[j] > 0$ **alors**
24: $Accept_i[j] \leftarrow Accept_i[j] - 1$;
25: **sinon** ▷ $Accept_i[j] = 0$
26: $\langle v_i, ts_i \rangle \leftarrow \langle v, ts \rangle$; $Accept_i[1..n] \leftarrow [2, \ldots, 2]$;
27: **envoyer** $(seq_i, \langle v_i, ts_i \rangle, seq)$ **à** p_j ;

L'algorithme 4.3 implémente un α-registre avec $\alpha = 2M - 1$ où $M = 2f - n + 2$.

Comme pour la simulation ABD, chaque valeur écrite v est associée à une unique estampille ts (ligne 8). Grâce au côté mono-écrivain du registre, chaque estampille ts est associée à une unique valeur v, et si on a $ts_1 < ts_2$, les valeurs associées v_1 et v_2 sont telles que l'opération Ecriture(v_1) précède l'opération Ecriture(v_2). Chaque processus stocke localement une paire de variables $\langle v_i, ts_i \rangle$ qui stocke la valeur la plus récente connue par p_i ainsi que son estampille. On dit qu'un processus p_i *accepte* une paire $\langle v, t \rangle$ quand p_i change son $\langle v_i, ts_i \rangle$ en $\langle v, t \rangle$ (ligne 26).

Les processus s'échangent en permanence des messages contenant la valeur la plus récente connue par l'expéditeur du message associée à son estampille. Cet algorithme est

ainsi *bruyant* dans le sens où, même sans opération en cours, les processus continuent d'envoyer et recevoir des messages indéfiniment. En plus de contenir $\langle v, ts \rangle$, les messages contiennent aussi deux numéros de séquence *seq* et *ancien_seq*, qui correspondent au numéro de séquence de l'expéditeur et au numéro de séquence du destinataire tel qu'il a été envoyé dans le message précédent. Plus précisément, chaque processus p_i stocke localement son propre numéro de séquence seq_i, et l'envoie (dans *seq*) avec chaque message. Chaque message (sauf le premier ligne 6) est envoyé suite à la réception d'un message, et le numéro de séquence *seq* contenu dans ce message reçu est alors retranscrit dans le message envoyé en tant que *ancien_seq* (lignes 18 puis 27). Ainsi, si un processus p_i reçoit un message contenant un $ancien\_seq = seq_i$, c'est que ce message est une réponse à un message que p_i avait envoyé alors qu'il avait déjà ce même numéro de séquence seq_i local. Ce système permet donc de savoir si une réponse obtenue est suffisamment *récente* par rapport à la dernière modification de seq_i. Enfin, ces numéros de séquence sont modifiés lors d'une opération d'Ecriture (ligne 8) et à chaque phase d'une opération de Lecture (ligne 13).

Les opérations d'Ecriture fonctionnent de manière similaire aux Ecriture de la simulation ABD, en attendant qu'un quorum Q_i d'au moins $n - f$ processus ait correctement *accepté* la nouvelle valeur $\langle v, ts \rangle$. En effet, la variable Q_i est réinitialisée au début de l'Ecriture puis ne contient que des processus p_j vérifiant $\langle v_j, ts_j \rangle = \langle v, ts \rangle$, comme garanti ligne 21.

La diffusion des valeurs est cependant bien différente de celle dans la simulation ABD. En particulier, un processus p_i qui reçoit un message contenant une valeur v plus récente que sa variable locale v_i ne l'*acceptera* pas nécessairement. Ceci dans le but de restreindre le nombre de valeurs obsolètes différentes pouvant être lues, et donc la valeur du α de la propriété de lectures α-bornées de cet α-registre. Plus précisément, une valeur v envoyée par p_j sera acceptée par p_i à la réception du 3ème message envoyé par p_j contenant une valeur plus récente que v_i, à condition que ces 3 messages aient été reçu depuis la dernière modification de v_i. Cela est réalisé à l'aide du tableau $Accept_i$, qui est initialisé à 2 et contient en permanence des valeurs dans $\{0, 1, 2\}$. Lorsque $Accept_i[j]$ contient 1 ou 2, le prochain message reçu par p_i de p_j, et contenant une valeur v plus récente que v_i, sera ignoré (pour ce qui est de la mise à jour de v_i), et $Accept_i[j]$ sera décrémenté (ligne 24). Si $Accept_i[j]$ vaut 0, alors la valeur v du prochain message de p_j sera acceptée, et l'ensemble du tableau $Accept_i$ sera réinitialisé à 2 (ligne 26). Ainsi, on restreint le nombre de valeurs pouvant être acceptées, sans pour autant bloquer la propagation des valeurs, grâce au système d'envoi systématique de réponse à réception d'un message. Le fonctionnement de ce tableau *Accept* est illustré par la figure 4.4.

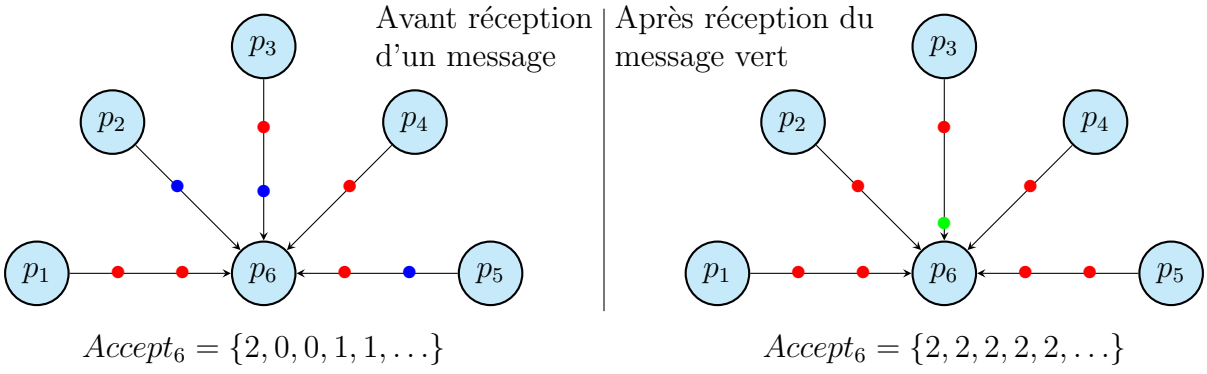

FIGURE 4.4 – Fonctionnement de *Accept*. En supposant qu'ils contiennent tous une valeur plus récente que celle de v_6, les points bleus représentent des messages pouvant être acceptés (selon l'exécution) et les rouges les messages qui ne peuvent pas être acceptés peu importe l'exécution (et donc non *utiles*)

De plus, la figure 4.5 permet de mieux comprendre le fait qu'un message de p_j (contenant une valeur plus récente) est accepté par p_i s'il est le 3ième tel message reçu par p_i depuis la réinitialisation de *Accept*.

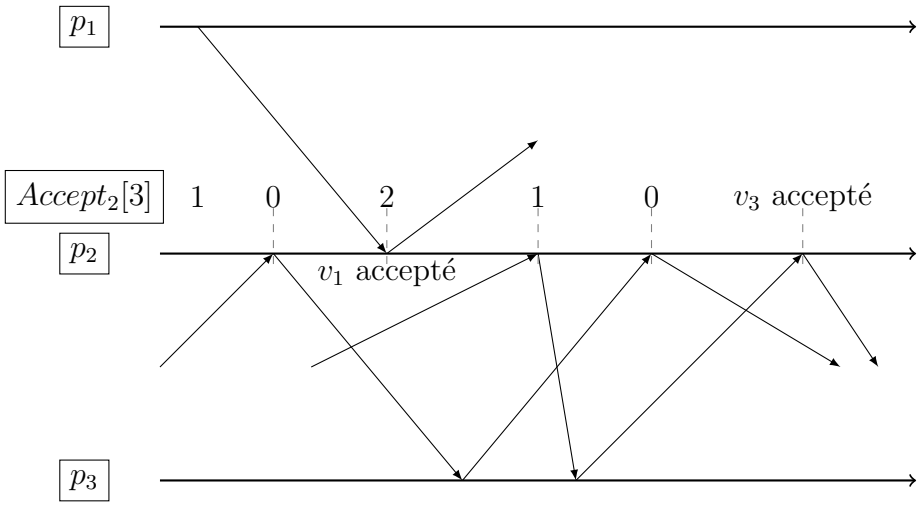

FIGURE 4.5 – Propriété de *Accept*. Les flèches obliques représentent les messages que s'échangent les processus, et on suppose qu'initialement $v_3 \succ v_1 \succ v_2$, et $Accept_2[1] = 0$

Une opération de Lecture par p_i est composée d'au plus $N = O(f^2)$ itérations. Chacune de ces itérations est identifiée par un numéro de séquence seq_i. Au début d'une itération, la paire $\langle v, ts \rangle$ stockée dans $\langle v_i, ts_i \rangle$ est copiée dans $\langle vl_i, tsl_i \rangle$, et Q_i et Ql_i sont réinitialisés à $\emptyset$. Ensuite, via les messages qui voyagent en continu d'un processus à l'autre, les ensembles Q_i et Ql_i vont être remplis de processus p_j tels que $v_j = vl_i$ et $v_j > vl_i$ respectivement. Une itération termine alors lorsqu'un quorum de $n - f$ processus contient des valeurs $v_j \geq vl_i$, c'est-à-dire quand $|Q_i \cup Ql_i| \geq n - f$. Enfin, l'opération termine (1) immédiatement si $|Q_i| \geq n - f$, c'est-à-dire qu'on dispose d'un quorum tel que chaque processus p_j de Q_i vérifie $v_j = vl_i$ à un moment durant cette itération ; ou (2) après N itérations. On démontrera que, pour chaque opération de Lecture exécutée sans Ecriture concurrente, cette opération termine par la condition (1).

Démonstrations.

On considère que le processus écrivain est p_n, et on note var_i^τ la valeur de la variable locale var_i (de p_i) à l'instant τ.

Comme il n'y a qu'un seul processus écrivain, les estampilles sont associées aux valeurs de manière unique, et chaque nouvelle estampille est strictement plus grande que la précédente (ligne 8). Ainsi, on peut dire qu'une valeur v est *plus récente* qu'une valeur v' si leurs estampilles associées vérifient $ts > ts'$, ce qu'on notera $\langle v', ts' \rangle \prec \langle v, ts \rangle$. On sait que pour chaque processus p_i, quand un couple de variable $\langle v_i, ts_i \rangle$ est modifié, il est remplacé par une valeur plus récente (lignes 8 et 26), d'où :

Observation 4.3.1. *Pour chaque processus p_i, et pour tout instant $\tau < \tau'$, $\langle v_i^\tau, ts_i^\tau \rangle \preceq \langle v_i^{\tau'}, ts_i^{\tau'} \rangle$.*

À chaque fois qu'un processus p_i reçoit un message de p_j, il lui répond par un autre message (lignes 18 et 27). De plus, initialement, chaque processus envoie un message à chacun des processus (ligne 6). On peut donc en conclure que des messages sont perpétuellement échangés entre p_i et p_j s'ils sont tous deux corrects.

Lemme 4.3.2. *Soient p_i, p_j deux processus corrects. p_i reçoit une infinité de messages de la part de p_j.*

Le lemme suivant montre que lorsqu'un processus correct apprend une nouvelle valeur, chaque autre processus correct finira par apprendre une valeur au moins aussi récente. Cela sert de base pour montrer que les opérations de Lecture (lemme 4.3.4) et d'Ecriture (lemme 4.3.5) appelées par des processus corrects terminent.

Lemme 4.3.3. *Soient p_i, p_j deux processus corrects. Si, à un moment, $\langle v_i, ts_i\rangle = \langle v, ts\rangle \neq \langle \perp, 0\rangle$, alors au bout d'un certain temps $\langle v, ts\rangle \preceq \langle v_j, ts_j\rangle$.*

Démonstration. D'après le lemme 4.3.2, le processus p_j reçoit une infinité de messages du processus p_i. Chacun de ces messages contient la valeur stockée dans $\langle v_i, ts_i\rangle$ quand le message est envoyé (ligne 27). Comme les estampilles stockées dans ts_i sont croissantes (observation 4.3.1), p_j reçoit une infinité de messages contenant $\langle v', ts'\rangle$ envoyés par p_i avec $\langle v, ts\rangle \preceq \langle v', ts'\rangle$. Supposons qu'un de ces messages soit reçu par p_j avec $\langle v_j, ts_j\rangle \prec \langle v, ts\rangle$ (sinon, le lemme est vrai). Donc, soit $Accept_j[i] > 0$ et est décrémenté (ligne 24), soit $Accept_j[i] = 0$ et $\langle v_j, ts_j\rangle$ est remplacé par $\langle v', ts'\rangle$, ce qui prouverait ce lemme.

Supposons par contradiction que ce lemme est faux, et donc que $Accept_j[i] > 0$ à chaque fois qu'un message de p_i est reçu par p_j. Comme $Accept_j[i]$ ne peut qu'être décrémenté à réception d'un tel message, ou réinitialisé à 2 lors du changement de $\langle v_j, ts_j\rangle$, et que p_j reçoit une infinité de messages de p_i, cela signifie que $Accept_j[i]$ est réinitialisé à 2 infiniment souvent. Or, cette réinitialisation n'a lieu que lors $\langle v_j, ts_j\rangle$ est remplacé par un $\langle v'', ts''\rangle \succ \langle v_j, ts_j\rangle$, c'est-à-dire une paire avec une estampille strictement plus grande. Comme les estampilles sont des entiers naturels, on peut en déduire que l'infinité de valeurs que prend ts_j finit par dépasser ts, et donc que, à partir d'un certain temps, $\langle v, ts_j\rangle \succeq \langle v, ts\rangle$. □

Lemme 4.3.4. *Soit p_i un processus correct. Chaque appel de l'opération Lecture() par p_i retourne.*

Démonstration. Supposons par contradiction qu'une opération de Lecture() L par p_i ne termine pas. D'après l'algorithme, cela signifie que p_i attend pour toujours à la ligne 14, car le nombre d'itérations ne peut pas dépasser N et donc la boucle extérieure ne peut pas être infinie. Donc, à partir d'un certain temps, on a $|Q_i \cup Ql_i| < n - f$ pour toujours.

Soient $\langle vl, tsl\rangle$ et s les valeurs respectivement stockées dans $\langle vl_i, tsl_i\rangle$ et seq_i immédiatement avant que p_i n'arrive à la ligne 14 de son attente infinie. On peut noter que les variables $\langle vl_i, tsl_i\rangle$ et seq_i ne sont plus modifiées une fois que p_i a commencé à attendre, et que Q_i et Ql_i ne sont plus réinitialisés à $\emptyset$ à partir de ce moment.

Soit p_j un processus correct. Comme p_i est correct et que $\langle vl, tsl\rangle$ est stocké dans $\langle v_i, ts_i\rangle$ à un moment, alors, d'après le lemme 4.3.3, au bout d'un certain temps, $\langle vl, tsl\rangle \preceq \langle v_j, ts_j\rangle$. D'après le lemme 4.3.2 et comme s est la dernière valeur de seq_i, p_j reçoit de p_i une infinité de messages $(s, \_, \_)$. Pour chacun de ces messages, p_j répond par un message $(\_, \_, s)$ (ligne 27). En particulier, p_j envoie une infinité de réponses $(\_, \langle v', ts'\rangle, s)$ avec $\langle v', ts'\rangle = \langle v_j, ts_j\rangle \succeq \langle vl, tsl\rangle$. Donc, d'après l'algorithme (lignes 19-20), p_j finit par être ajouté à l'ensemble $Q_i \cup Ql_i$. Comme cela est vrai pour tout processus correct p_j, et que $Q_i \cup Ql_i$ est croissant dans le temps (par la relation $\subseteq$) car non réinitialisé, au bout d'un certain temps $|Q_i \cup Ql_i| \geq n - f$ (car au moins $n - f$ processus sont corrects). Donc l'attente de p_i ligne 14 se termine, et donc l'opération de Lecture() aussi. □

Lemme 4.3.5. *Supposons que l'écrivain p_n soit un processus correct. Alors, chaque appel à l'opération Ecriture() par p_n termine.*

Démonstration. Supposons par contradiction qu'une opération d'Ecriture() E ne termine pas. Soit v le paramètre de cette opération, et ts l'estampille associée (ligne 8). Soit s la valeur stockée dans seq_n après la ligne 8, etsoit p_i un processus correct. Au début de E, $\langle v_n, ts_n\rangle$ prend pour valeur $\langle v, ts\rangle$. Comme p_n est supposé correct, d'après le lemme 4.3.3,

au bout d'un certain temps $\langle v, ts\rangle \preceq \langle v_i, ts_i\rangle$. De plus, comme les nouvelles estampilles ne sont introduites que lors d'opération d'Ecriture(), on peut déduire du fait que E ne termine pas que ts est la plus grande estampille (car la plus récente). Donc, au bout d'un certain temps, $\langle v_i, ts_i\rangle$ n'est plus modifié et contient $\langle v, ts\rangle$ pour toujours. D'après le lemme 4.3.2, p_n reçoit une infinité de message de p_i, sous la forme $(s, \langle v, ts\rangle, \_)$. Donc, comme $\langle vl, tsl\rangle = \langle v, ts\rangle$, on a au bout d'un certain temps $p_i \in Q_i$ (ligne 21). Cela est valide pour chacun des au moins $n - f$ processus corrects, et donc au bout d'un certain temps $|Q_i| \geq n - f$ (ligne 9). Donc l'opération E termine, ce qui démontre ce lemme. □

Propriété 4.3.6. *Cet algorithme vérifie la propriété de Terminaison des α-registres.*

On peut remarquer que la valeur retournée par une opération de Lecture() est celle stockée dans v_i, et que ces valeurs ne sont modifiées qu'après réception d'un message ou lors d'une Ecriture(). Dans le premier cas, la nouvelle valeur est une valeur qui était stockée dans un v_j, et dans le second il s'agit du paramètre v de l'opération d'Ecriture() en question. Enfin, initialement, les v_i contiennent $\perp$. Donc, à tout instant, les v_i ne contiennent que $\perp$ ou une valeur v telle qu'une opération Ecriture(v) a été appelée. On a donc :

Propriété 4.3.7. *Cet algorithme vérifie la propriété de Non-trivialité des α-registres.*

D'après l'observation 4.3.1, on sait que les valeurs stockées dans v_i ne peuvent être remplacées que par des valeurs *plus récentes*. Comme chaque opération de Lecture() retourne la valeur stockée dans v_i, on sait que deux Lecture() successives par un même processus p_i retourneront deux valeurs v et v' telles que la seconde valeur retournée est plus récente que la première, d'où :

Propriété 4.3.8. *Cet algorithme vérifie la propriété de Monotonie des Lectures des α-registres.*

De même, comme après chaque Ecriture(), la variable v_n contient la dernière valeur écrite, on vérifie bien :

Propriété 4.3.9. *Cet algorithme vérifie la propriété de Visibilité des Ecritures des α-registres.*

Enfin, grâce au lemme 4.3.3, ainsi qu'aux propriétés précédentes, on a :

Propriété 4.3.10. *Cet algorithme vérifie les propriétés de Cohérence Finale et de Propagation des α-registres.*

Il reste donc à démontrer que cet algorithme vérifie la propriété de Lecture α-bornée pour un certain α.

On considère un intervalle de temps $\mathcal{I}$ et $\mathcal{L} = \{L_1, \ldots, L_m\}$ l'ensemble des opérations de Lecture() incluses dans $\mathcal{I}$ (c'est-à-dire appelées après le début de $\mathcal{I}$ et retournant avant la fin de $\mathcal{I}$). Soit u_i la valeur retournée par l'opération L_i, et soit $V_L = \{u_1, ..., u_m\}$. On note $\mathcal{E}$ l'ensemble des Ecriture() concurrentes à $\mathcal{I}$ (c'est-à-dire commençant avant la fin de $\mathcal{I}$ et ne retournant pas avant le début de $\mathcal{I}$), et V_E l'ensemble des paramètres de ces opérations. Enfin, on note $V_v = V_L \setminus V_E$ l'ensemble des *vieilles* valeurs lues.

On souhaite montrer que :

Lemme 4.3.11. $|V_v| \leq 2M - 1$ *où* $M = \max(1, 2f - n + 2)$.

Les valeurs de V_v sont présentes dans le système avant $\mathcal{I}$, et cela peut être sous deux formes différentes : elles peuvent être stockées localement dans des variables v_i des processus, ou contenues dans des messages qui n'ont pas encore été reçus au début de $\mathcal{I}$. Ainsi, en notant τ_0 le début de $\mathcal{I}$, on définit :

— $Mess_{j\to i}$ l'ensemble des messages envoyés par p_j à p_i mais pas encore reçus à l'instant τ_0, présents dans les canaux de communication.
— V_p= $\{v \notin V_E : \exists p_i, \langle v_i^{\tau_0}, ts_i^{\tau_0}\rangle = \langle v, ts\rangle\}$
— V_m= $\{v \notin V_E \cup V_p : \exists p_i, p_j$ tel que $\exists(\_, \langle v, ts\rangle, \_) \in Mess_{j\to i}\}$
— v_d la valeur écrite lors de la dernière opération d'Ecriture() qui précède $\mathcal{I}$ ($v_d = \perp$ s'il n'y a aucune opération d'Ecriture précédent $\mathcal{I}$)

On a alors $V_p \cup V_m$ qui représente l'ensemble des valeurs du système à l'instant τ_0 qui ne font pas partie de V_E, et donc $V_v = V_L \cap (V_p \cup V_m)$. Si $v_d = \perp$, alors $V_v \subseteq \{\perp\}$, et la propriété de Lecture α-bornée est alors vérifiées avec $\alpha = 1$. On considérera donc un intervalle $\mathcal{I}$ tel que $v_d \neq \perp$. En particulier, on a $\forall v \in V_v$, $\langle v, ts\rangle \preceq \langle v_d, ts_d\rangle$. Ces ensembles sont représentés dans la figure 4.6, y compris U qui sera introduit prochainement.

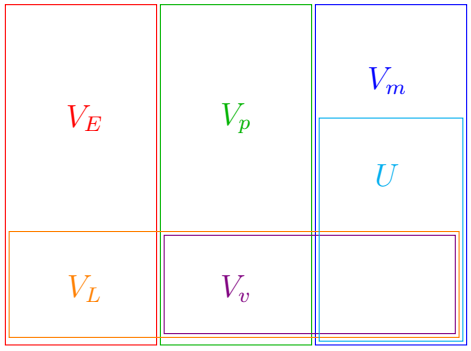

FIGURE 4.6 – Visualisation des notations utilisées dans les lemmes suivants

On observe tout d'abord que $|V_p| \leq f + 1$ (corollaire 4.3.13). Cela vient du fait qu'un quorum Q_d d'au moins $n-f$ processus doit avoir accepté la dernière valeur écrite v_d avant $\mathcal{I}$ lors de l'opération d'Ecriture correspondante E_d (lemme 4.3.12). Comme les valeurs stockées dans les variables locales $\langle v_i, ts_i\rangle$ sont croissantes, on peut en déduire que, à tout instant dans $\mathcal{I}$, pour tout processus $p_i \in Q_d$, $v_i \in \{v_d\} \cup V_E$.

Lemme 4.3.12. *Soit v_d la valeur écrite par la dernière opération d'Ecriture() précédant $\mathcal{I}$, et ts_d l'estampille associée. Il existe un ensemble Q_d d'au moins $n - f$ processus tel que, pour tout processus $p_i \in Q_d$, à tout instant dans $\mathcal{I}$, $v_i \in \{v_d\} \cup V_E$ et $ts_i \geq ts_d$.*

Démonstration. Soit E_d la dernière Ecriture précédant $\mathcal{I}$. Quand cette opération termine, p_n a reçu un message $(\_, \langle v_d, ts_d\rangle, \_)$ de chacun des processus dans un ensemble Q_n d'au moins $n - f$ processus (ligne 9). Donc, pour chaque processus $p_i \in Q_n$, à un moment durant E_d, on a $\langle v_i, ts_i\rangle = \langle v_d, ts_d\rangle$. Comme les valeurs stockées dans les $\langle v_i, ts_i\rangle$ sont croissantes (observation 4.3.1), on a, à tout instant dans $\mathcal{I}$ (qui a lieu après E_d), $ts_i \geq ts_d$. Et comme les seules opérations d'Ecriture() appelées après E_d et avant la fin de $\mathcal{I}$ ont pour paramètre une valeur de V_E, toutes les valeurs possibles de v_i sont dans $\{v_d\} \cup V_E$. □

Corollaire 4.3.13. $|V_p| \leq f + 1$

Démonstration. D'après le lemme 4.3.12, au début de $\mathcal{I}$, pour chaque processus $p_i \in Q_d$, $v_i \in \{v_d\} \cup V_E$. Comme $V_E \cap V_p = \emptyset$ et comme $|Q_d| \geq n-f$, $|V_p| \leq n-(n-f)+1 = f+1$. □

Considérons deux processus p_j et p_i. D'après l'algorithme, p_j envoie un message à p_i à chaque fois qu'il reçoit un message de ce dernier (lignes 18 à 27). Comme initialement, p_i et p_j envoient tous deux un message à l'autre, on peut en déduire qu'à tout instant, au plus deux messages ont été envoyés par p_j à p_i mais n'ont pas encore été reçu.

Observation 4.3.14. *Pour chaque paire de processus p_i, p_j, $|Mess_{j\to i}| \leq 2$.*

Soit $U \subseteq V_m$ l'ensemble des valeurs qui ne sont pas stockées localement sur des processus au début de $\mathcal{I}$ mais qui sont stockées sur au moins un processus à un moment durant $\mathcal{I}$. C'est-à-dire $U = \{v \in V_m : \exists \tau \in \mathcal{I}$ et $\exists p_i$ tels que $v_i^\tau = v\}$. En particulier, on a $V_m \cap V_l = U \cap V_L$, car toute valeur lue doit être stockée localement par un processus à un moment de la Lecture.

On borne alors la taille de U par f (lemme 4.3.15). Cette borne est un élément important pour montrer que, pour toute vieille valeur v lue durant $\mathcal{I}$, il existe un ensemble d'au moins $n - f$ processus p_j tels que $v_j = v$ à un moment durant $\mathcal{I}$ (lemme 4.3.16).

Lemme 4.3.15. $|U| \leq f$

Démonstration. Soit $u \in V_m$ et ts son estampille. Supposons qu'à un certain instant $\tau \in \mathcal{I}$, $\langle v_x^\tau, ts_x^\tau \rangle = \langle u, ts \rangle$ pour un processus p_x. Au début de $\mathcal{I}$, aucun processus ne stocke localement u ($\forall p_j, v_j^{\tau_0} \neq u$), mais un message contenant u a été envoyé et pas encore reçu. Soit p_i le premier processus qui modifie sa paire $\langle v_i, ts_i \rangle$ pour prendre pour valeur $\langle u, ts \rangle$ (ligne 26). Comme $u \notin V_p$, cela arrive lorsque p_i reçoit un message $m = (\_, \langle u, ts \rangle, \_)$ envoyé à p_i avant $\mathcal{I}$. Et donc il existe un processus p_j tel que $m \in Mess_{j \to i}$.

Considérons une autre valeur $u' \neq u$ qui, de la même manière que u, (1) est dans V_m et (2) à un instant $\tau' \in \mathcal{I}$ vérifie $v_{x'}^{\tau'} = u'$ pour un processus $p_{x'}$. Soit $p_{i'}$ le premier processus qui modifie son $v_{i'}$ en prenant u' pour valeur. Cela arrive à la réception d'un message $m' \in Mess_{j' \to i'}$.

Supposons par contradiction que $p_i = p_{i'}$. Sans perdre en généralité, on supposera que p_i change sa valeur v_i pour u en premier, puis plus tard pour u'. D'après l'algorithme, immédiatement après que $\langle v_i, ts_i \rangle$ soit modifié, le tableau $Accept_i$ est réinitialisé à $[2, \ldots, 2]$ (ligne 26). Comme les canaux de communication sont FIFO, tout message reçu de $p_{j'}$ par p_i durant $\mathcal{I}$ et avant que m' ne soit reçu est contenu dans $Mess_{j' \to i}$. Donc, entre la réception de m et celle de m', au plus $|Mess_{j' \to i}| - 1$ messages de $p_{j'}$ sont reçus par p_i. Comme $|Mess_{j' \to i}| \leq 2$ (observation 4.3.14), cela signifie que $Accept_i[j'] > 0$ lorsque m' est reçu par p_i (car il n'est décrémenté qu'au plus une fois par message reçu de $p_{j'}$ par p_i, ligne 24). Donc v_i reste inchangé lorsque m' est reçu (lignes 24 à 26) , ce qui amène une contradiction.

Finalement, on peut noter que ni p_i ni $p_{i'}$ ne peuvent être contenus dans Q_d, puisque pour chaque processus $p_y \in Q_d$, v_y ne peut être remplacé que par une valeur de V_E (lemme 4.3.12). Comme $|Q_v| \geq n - f$ et que chaque valeur $u \in U$ doit être initialement reçue par un processus différent pour être acceptée, on peut conclure que $|U| \leq f$. □

Lemme 4.3.16. *Soit $L \in \mathcal{L}$ une opération de Lecture() et v sa valeur de retour. Alors, $v \in V_E$ ou il existe un ensemble Q_L d'au moins $n - f$ processus tel que $\forall p_i \in Q_L$, il existe un instant τ pendant l'exécution de L tel que $v_i^\tau = v$.*

Démonstration. Supposons par contradiction que ce lemme est faux, et donc en particulier $v \notin V_E$. Soient ts l'estampille associée à v, et p_i le processus qui a appelé l'opération L. La boucle **répéter** (lignes 12 à 16) exécutée par p_i termine après N itérations ou quand $|Q_i| \geq n - f$. Dans le second cas, cela signifie que p_i reçoit un message contenant $\langle v, ts \rangle$ de chacun des processus $p_j \in Q_i$ (ligne 21). Donc, pour chacun de ces processus, il y a un moment durant l'exécution de L pour lequel $\langle v_j, ts_j \rangle = \langle v, ts \rangle$, car chaque message reçu qui modifie Q_i a été envoyé après le début de L, comme le garanti le numéro de séquence seq_i (ligne 19). Comme on suppose que le lemme est faux, ce cas ne peut pas arriver.

Donc, p_i exécute N itérations de la boucle **répéter**. Dans chaque itération associée à un numéro de séquence $seq_i = s$, p_i doit recevoir au moins un message de la forme $(\_, \langle u^s, ts^s \rangle, s)$ avec $\langle v_i^s, ts_i^s \rangle \prec \langle u^s, ts^s \rangle$, où $\langle v_i^s, ts_i^s \rangle$ est la valeur de $\langle v_i, ts_i \rangle$ au début de cette itération (c'est-à-dire la valeur stockée dans $\langle vl_i, tsl_i \rangle$ pour cette itération). En effet,

pour terminer cette itération sans avoir $|Q_i| \geq n - f$, p_i doit vérifier $|Ql_i| \geq 1$, et donc recevoir un message avec une valeur strictement plus récente que $\langle v_i^s, ts_i^s \rangle$.

Examinons ce qui arrive lorsque p_i reçoit un message m contenant un tel $\langle u^s, ts^s \rangle \succ \langle v_i^s, ts_i^s \rangle$ de p_j. Si $ts_i < ts^s$, alors $Accept_i[j]$ est décrémenté ou, si il vaut 0, $\langle v_i, ts_i \rangle$ prend pour valeur $\langle u^s, ts^s \rangle$. Sinon, à la réception du message, $ts_i \geq ts^s > ts_i^s$, et donc la valeur de $\langle v_i, ts_i \rangle$ a été remplacé par un certain $\langle v', ts' \rangle \succ \langle v_i^s, ts_i^s \rangle$ entre le début de cette itération et la réception de ce message m. En résumé, à chaque itération, la valeur de $\langle v_i, ts_i \rangle$ est modifiée et/ou le compteur $Accept_i[j]$ est décrémenté pour un certain p_j.

Considérons un bloc b de $N_1 = 2(n-(n-f-1))+1 = 2f+3$ itérations consécutives. Supposons que la valeur de $\langle v_i, ts_i \rangle$ n'est pas modifiée au cours de ce bloc b de N_1 itérations. Durant chacune de ces itérations p_i reçoit des messages d'au moins $n - f$ processus différents, dont au moins un contient un $\langle u^s, ts^s \rangle \succ \langle v_i^s, ts_i^s \rangle$. Donc, au cours de ces N_1 itérations, au moins 3 itérations voient un même processus p_j envoyer un tel message, car une fois qu'un processus envoie un tel message, chaque message suivant qu'il enverra contiendra aussi une valeur strictement plus récente que $\langle v_i, ts_i \rangle$. Durant chacune de ces 3 itérations, comme $ts^s > ts_i$, $Accept_i[j]$ est décrémenté, car on a supposé que $\langle v_i, ts_i \rangle$ n'était pas modifié, donc le cas $Accept_i[j] = 0$ ne doit pas arriver. Or, $Accept_i[j]$ n'est modifié qu'en étant décrémenté, ou réinitialisé à 2 lors de la modification de $\langle v_i, ts_i \rangle$. Comme $\langle v_i, ts_i \rangle$ n'est pas modifié, $Accept_i[j]$ ne peut qu'être décrémenté, et donc à la 3ème itération, on a bien $Accept_i[j] = 0$, ce qui amène une contradiction. Donc, durant chaque bloc de N_1 itérations successives, $\langle v_i, ts_i \rangle$ est modifié au moins une fois.

Maintenant, on sait que l'opération de Lecture() est incluse dans l'intervalle $\mathcal{I}$ où toute nouvelle valeur écrite appartient à V_E. Les valeurs possibles que peut prendre $\langle v_i, ts_i \rangle$ au cours de cette opération (y compris initialement) sont dans $V_p \cup U \cup V_E$, par définition de ces ensembles. Hors, d'après le corollaire 4.3.13 et le lemme 4.3.15, $|V_p \cup U| \leq 2f+1 = N_2$. On sait aussi que chaque valeur de V_E est strictement plus récente que chaque valeur de $V_p \cup U$. Comme les valeurs successives de $\langle v_i, ts_i \rangle$ sont croissantes (observation 4.3.1), on en déduit qu'en $N_1 * N_2$ itérations, v_i a changé de valeur au moins N_2 fois, et a donc atteint une valeur dans V_E. Donc, après une itération supplémentaire, qui arrive nécessairement car $N = N_1 * N_2 + 1$, la valeur de vl_i est dans V_E, et donc la Lecture retourne une valeur de V_E. Cela contredit l'hypothèse que $v \notin V_E$, et prouve ce lemme. □

Finalement, on borne plus précisément le nombre de valeurs de V_p pouvant être lues au cours de $\mathcal{I}$ (lemme 4.3.17) et le nombre de valeurs de V_m pouvant être lues pendant $\mathcal{I}$ (lemme 4.3.18). On peut alors simplement déduire une borne sur le nombre de vieilles valeurs pouvant être lues.

Lemme 4.3.17. $|V_p \cap V_L| \leq M = 2f - n + 2$

Démonstration. Ce lemme est vrai si $|V_p| \leq 2f - n + 2$. Supposons donc que $|V_p| = x > 2f - n + 2$, et notons $u_1, \ldots, u_x$ les valeurs de V_p ordonnées de la plus ancienne à la plus récente. Ainsi, en notant t_i l'estampille associée à u_i, on a $\forall i \in [1, x-1], t_i < t_{i+1}$. Soit $m = x - (2f - n + 2) > 0$.

D'après le lemme 4.3.12, il y a un ensemble Q_d d'au moins $n-f$ processus p_i tels que, à tout instant dans $\mathcal{I}$, $v_i \notin V_p \setminus \{v_d\}$. Or, par définition de v_d, on sait que $\langle u_x, t_x \rangle \preceq \langle v_d, ts_d \rangle$. Par définition de V_p, pour au moins $(x-1) - (m+1) + 1 = 2f - n + 1$ processus $p_j \notin Q_d$, la valeur de v_j au début de $\mathcal{I}$ est dans l'ensemble $\{u_{m+1}, \ldots, u_{x-1}\}$. Donc, au plus $n-((n-f)+(2f-n+1)) = n-f-1$ processus p_j ont une valeur v_j dans $\{u_1, \ldots, u_m\}$ au début de $\mathcal{I}$.

D'après le lemme 4.3.16, pour qu'une valeur $u \in \{u_1, \ldots, u_m\}$ soit retournée par une opération de Lecture(), chaque processus p_j dans un ensemble Q d'au moins $n-f$ processus doit stocker u dans sa variable v_j à un moment durant l'opération et donc durant $\mathcal{I}$, car $u \notin$

V_E. Comme la variable v_j d'un processus ne peut être modifiée que pour prendre une valeur plus récente (observation 4.3.1), Q ne peut contenir que des processus dont les valeurs au début de $\mathcal{I}$ étaient plus anciennes que u, et donc des processus dans $\{u_1, \ldots, u_m\}$. Comme il n'y a qu'au plus $n - f - 1$ tels processus, aucune valeur $u \in \{u_1, \ldots, u_m\}$ ne peut être lue. Donc, $V_p \cap V_L \subseteq \{u_{m+1}, \ldots, u_x\}$, d'où on a $|V_p \cap V_L| \leq 2f - n + 2$. □

Lemme 4.3.18. $|V_m \cap V_L| \leq M - 1 = 2f - n + 1$

Démonstration. Comme dans la preuve du lemme 4.3.17, soient $u_1, \ldots, u_x$ les valeurs de $V_m \cap V_L$, ordonnées de la plus ancienne à la plus récente. Comme la valeur u_1 est retournée par une opération de Lecture() et que $u_1 \notin V_E$, il existe un ensemble Q_1 de $n-f$ processus qui ont $v_j = u_1$ à un moment durant cette opération et donc durant $\mathcal{I}$.

Soit $u' \in V_m \cap V_L$, $u' \neq u_1$ (si un tel u' n'existe pas, le lemme est vrai), associé à l'estampille t'. On démontre que le premier processus p_i qui modifie son v_i pour prendre pour valeur u' n'appartient pas à Q_1. Supposons par contradiction que $p_i \in Q_1$, et donc que $Mess_{k \to i}$ contienne un message $(\_, \langle u', t' \rangle, \_)$. Comme $\langle u_1, t_1 \rangle \notin V_p$, $\langle v_i, ts_i \rangle$ doit être modifié pour contenir $\langle u_1, t_1 \rangle$ à un moment dans $\mathcal{I}$. Quand cela arrive, $Accept_i[k]$ est réinitialisé à 2 (ligne 26). Par l'observation 4.3.1, $\langle v_i, ts_i \rangle = \langle u', t' \rangle$ ne peut arriver qu'après que $\langle v_i, ts_i \rangle = \langle u_1, t_1 \rangle$, car $t' > t_1$, donc seulement après que $Accept_i[k]$ ait été réinitialisé à 2. Comme les canaux de communication sont FIFO, les messages dans $Mess_{k \to i}$ sont reçus par p_i avant tout message envoyé de p_k à p_i pendant $\mathcal{I}$. Enfin, comme $|Mess_{k \to i}| \leq 2$ (observation 4.3.14), et comme le compteur $Accept_i[k]$ ne peut décroître que lorsque p_i reçoit un message de p_k, on sait que $Accept_i[k] > 0$ lorsque le message de p_k contenant $\langle u', t' \rangle$ est reçu par p_i. Donc ce message ne peut pas modifier la valeur de $\langle v_i, ts_i \rangle$ (ligne 26), ce qui est une contradiction.

Donc $p_i \notin Q_1$. On peut aussi noter que $p_i \notin Q_d$ et que $Q_1 \cap Q_d = \emptyset$, car les processus de Q_d ne peuvent stocker dans v_j que des valeurs plus récentes ou égales à v_d et que $v_d \succ u' \succ u_1$ (lemme 4.3.12). Donc, pour chaque valeur $u_j \in (V_m \cap V_L) \setminus \{u_1\}$, le premier processus durant $\mathcal{I}$ tel que $v_i = u_j$ n'appartient pas à $Q_1 \cup Q_d$. Enfin, comme lors de la preuve du lemme 4.3.15, pour tout $u_i \neq u_j \in V_m \cap V_L$, les premiers processus à obtenir $v = u_i$ et $v = u_j$ sont distincts. Donc, $|V_m \cap V_L| \leq n - |Q_1| + 1 - |Q_d| \leq n - (n - f) + 1 - (n - f) = 2f - n + 1$. □

Toute valeur retournée lors d'une Lecture() contenue dans l'intervalle $\mathcal{I}$ appartient à $V_p \cup V_m \cup V_E$. Comme $|V_p \cap V_L| \leq 2f - n + 2$ (lemme 4.3.17), et $|V_m \cap V_L| \leq 2f - n + 1$ (lemme 4.3.18), on a $|V_v| = |V_L \cap (V_p \cup V_m)| \leq 2f - n + 2 + 2f - n + 1 = 2M - 1$. Cela prouve le lemme 4.3.11.

Finalement, on peut déduire que :

Théorème 4.3.19. *L'algorithme 4.3 implémente un α-registre avec $\alpha = 2M - 1$.*

4.3.2 Version silencieuse

On présente maintenant un autre algorithme d'implémentation d'α-registre. Cette version est silencieuse, dans le sens où, contrairement à la version précédente, des messages ne circulent pas en permanence, et lorsque les opérations prennent fin, les messages arrêtent d'être envoyés après un temps. De plus, cette version est plus rapide en terme d'itérations lors des opérations de Lecture, car le nombre maximal de telles itérations est $O(f)$ au lieu de $O(f^2)$.

Algorithme 4.4 Algorithme silencieux d'implémentation d'α-registre (code exécuté par p_i)

1: **Initialisation**

2: $seq_i \leftarrow 1$; $\langle v_i, ts_i \rangle \leftarrow \langle \perp, 0 \rangle$; $\langle vl_i, tsl_i \rangle \leftarrow \langle \perp, 0 \rangle$;
3: $Ql_i \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$;
4: $cles_i[1..n] \leftarrow [0, \ldots, 0]$;
5: $cles_i[i] \leftarrow 1$;
6: **Procédure** ECRITURE(v)
7: $\langle v_i, ts_i \rangle \leftarrow \langle v, ts_i + 1 \rangle$; $\langle vl_i, tsl_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $Q_i \leftarrow \emptyset$;
8: **pour chaque** processus p_j **faire**
9: **envoyer** (Requete, seq_i, $\langle v_i, ts_i \rangle$, 0, $cles_i[i]$, $cles_i[j]$) **à** p_j ;
10: **attendre** jusqu'à ce que $|Q_i| \geq n - f$;
11: $seq_i \leftarrow seq_i + 1$;
12: **Fonction** LECTURE( )
13: $n\_iter \leftarrow 0$;
14: **répéter**
15: $\langle vl_i, tsl_i \rangle \leftarrow \langle v_i, ts_i \rangle$; $Ql_i \leftarrow \emptyset$; $Q_i \leftarrow \emptyset$;
16: **pour chaque** processus p_j **faire**
17: **envoyer** (Requete, seq_i, $\langle v_i, ts_i \rangle$, 0, $cles_i[i]$, $cles_i[j]$) **à** p_j ;
18: **attendre** jusqu'à ce que $|Ql_i \cup Q_i| \geq n - f$;
19: $n\_iter \leftarrow n\_iter + 1$; $seq_i \leftarrow seq_i + 1$;
20: **jusqu'à ce que** $(|Q_i| \geq n - f)$ ou $(n\_iter \geq N)$ ▷ $N = 2f + 2$
21: **retourner** vl_i ;
22: **Quand** le message (Reponse, seq, $\langle v, ts \rangle$, $ancien\_seq$, sa_cle, $vieille\_cle$) **est reçu** de p_j
23: $cles_i[j] \leftarrow sa\_cle$;
24: **si** $ts > ts_i$ **alors**
25: **si** $vieille\_cle = cles_i[i]$ **alors**
26: $\langle v_i, ts_i \rangle \leftarrow \langle v, ts \rangle$; $cles_i[i] \leftarrow cles_i[i] + 1$;
27: **si** $ancien\_seq = seq_i$ **alors**
28: **si** $ts > tsl_i$ **alors** $Ql_i \leftarrow Ql_i \cup \{p_j\}$;
29: **si** $ts = tsl_i$ **alors** $Q_i \leftarrow Q_i \cup \{p_j\}$;
30: **si** $ts < tsl_i$ **alors**
31: **envoyer** (Requete, seq_i,$\langle v_i, ts_i \rangle$, 0, $cles_i[i]$, $cles_i[j]$) **à** p_j ;
32: **Quand** le message (Requete, seq, $\langle v, ts \rangle$, $ancien\_seq$, sa_cle, $vieille\_cle$) **est reçu** de p_j
33: $cles_i[j] \leftarrow sa\_cle$;
34: **si** $ts > ts_i$ **alors**
35: **si** $vieille\_cle = cles_i[i]$ **alors**
36: $\langle v_i, ts_i \rangle \leftarrow \langle v, ts \rangle$; $cles_i[i] \leftarrow cles_i[i] + 1$;
37: **envoyer** (Reponse, seq_i, $\langle v_i, ts_i \rangle$, seq, $cles_i[i], cles_i[j]$) **à** p_j ;

L'algorithme 4.4 fonctionne sur le même principe général que le précédent (4.3). La différence entre les deux vient du système de communication, car là où le premier algorithme envoyait des messages en permanence, celui-ci n'envoie de messages que dans le cadre d'une opération Lecture ou Ecriture. Sans rentrer dans les détails, au cours de chaque opération, les deux algorithmes fixent Q_i et Ql_i à $\emptyset$, puis attendent que $|Q_i| \geq n-f$ ou $|Q_i \cup Ql_i| \geq n-f$, ce qui arrive grâce à des communications entre les processus. Dans ce nouvel algorithme, un message de type Requete est donc envoyé à tous les processus après avoir fixé les Q à $\emptyset$, afin d'initier une communication. Le système de numéro de séquence *seq* fonctionne comme pour l'algorithme précédent, si ce n'est qu'il est incrémenté *après* le test sur la taille des Q, au lieu d'avant. Il permet toujours d'identifier la communication en

cours, mais permet en plus de déterminer que la communication précédente est terminée, le début d'une telle étape étant marqué par l'envoi d'un premier message de Requete.

Un processus qui reçoit un message de Requete renvoie systématiquement un message de Reponse. Mais lorsqu'un processus reçoit un message de type Reponse, il ne lui répond (par une Requete) qu'à condition que la communication en cours soit toujours d'actualité (identifié par *seq*), et que la valeur contenue dans le message soit obsolète ($ts < tsl_i$). Cette particularité fait que, si toute opération est terminée, les processus recevant un message de type Reponse n'enverront pas de message de retour à l'expéditeur. Ainsi, hors opération, les messages envoyés mais pas encore reçus vont tendre à disparaître, car les Requetes laissent place à des Reponses qui, reçues, n'engendrent pas d'autres messages. On remarque donc bien que cet algorithme est *silencieux*.

Le système de clés avec les variable $cles_i$ permet de remplacer le tableau *Accept* de l'algorithme précédent, en limitant le nombre de messages non délivrés qui sont susceptibles de modifier l'état du destinataire. Ainsi, tout message envoyé par p_i à p_j contient, dans le champ *vielle_cle*, la valeur de $cles_i[j]$, qui est mise à jour à réception par p_i d'un message de p_j. Cette variable est supposée contenir la dernière valeur de $cles_j[j]$. L'idée générale est alors la suivante : un processus p_i ne met à jour son $\langle v_i, ts_i \rangle$ à réception d'un message avec un $\langle v, ts \rangle$ plus récent que si ce message contient la valeur à jour de $cles_i[i]$. Ensuite, après mise à jour de $\langle v_i, ts_i \rangle$, la valeur de $cles_i[i]$ est incrémentée, de manière à invalider les messages qui n'ont pas encore été reçus, et qui seront alors considérés comme obsolètes. Enfin, on peut noter que $cles_i[j]$ n'est utile que pour la communication entre p_i et p_j, donc les indices j ainsi utilisés n'ont pas besoin d'être cohérents d'un processus à l'autre, mais servent plutôt pour p_i à différencier ses interlocuteurs. Le système de clés est illustré dans la figure 4.7. Si p_i ne modifie pas son $cles_i[i]$ à réception d'un message de p_j, cela signifie que la valeur de ce message n'a pas été *acceptée*, et que p_i n'a pas modifié son v_i.

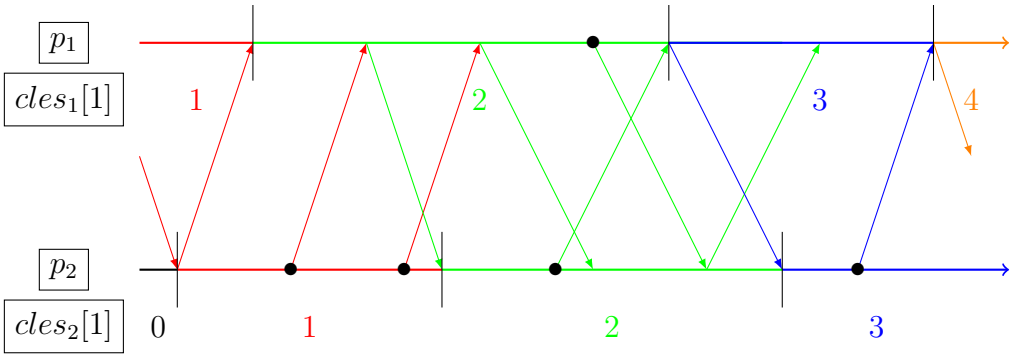

FIGURE 4.7 – Fonctionnement des *cles*. La couleur des messages indique la valeur contenue dans *vieille_cle* (de p_2 à p_1) ou *sa_cle* (inversement). Les points représentent l'envoi de message de type Requete

Afin de démontrer le bon fonctionnement de cet algorithme, on commence par remarquer que cet algorithme a lui aussi la propriété suivante :

Observation 4.3.20. *Pour chaque processus p_i, et pour tout instant $\tau < \tau'$, $\langle v_i^\tau, ts_i^\tau \rangle \preceq \langle v_i^{\tau'}, ts_i^{\tau'} \rangle$.*

D'où on a, comme précédemment :

Propriété 4.3.21. *Cet algorithme vérifie la propriété de Non-trivialité, de Monotonie des Lectures, et de Visibilité des Ecritures des α-registres.*

On peut aussi remarquer, d'après les lignes 32–37 :

Observation 4.3.22. *Si p_i et p_j sont corrects, et que p_i envoie à p_j un message (Requete, $seq, *, *, *, *$), p_i finira par recevoir un message (Reponse, $*, *, seq, *, *$) de p_j.*

On remarque que les valeurs de $cles_i[j]$ ne sont modifiées que pour prendre une valeur contenue dans le champ *sa_cle* d'un message envoyé par p_j à p_i. De plus, les valeurs de $cles_i[i]$ ne sont modifiées que lors d'incrémentations associées à une modification de $\langle v_i, ts_i \rangle$ pour une valeur plus grande (lignes 26 et 36). D'où on a :

Lemme 4.3.23. *Si p_i et p_j sont corrects, et que p_i envoie un message (Requete, $seq, \langle v, ts \rangle, *, *, *$) à p_j, alors p_i reçoit une réponse (Reponse, $*, \langle v', ts' \rangle, seq, *, *$). Si on a $seq_i = seq$, alors soit $ts' \geq tsl_i$, soit p_i recevra plus tard un autre message (Reponse, $*, \langle v'', ts'' \rangle, seq, *, *$) de la part de p_j, avec $ts'' > ts'$.*

Démonstration. Supposons que, quand p_i reçoit une réponse (Reponse, $*, \langle v', ts' \rangle, seq, sa\_cle, *$) de p_j, $seq_i = seq$ et $ts' < tsl_i$. En particulier, par définition de tsl_i, on a $tsl_i \leq ts$, d'où $ts' < ts$. Donc (ligne 30), p_i envoie un message (Requete, $seq, \langle v_i, ts_i \rangle, *, *, c$) à p_j, avec $c = sa\_cle$ reçu via le message Reponse en question. Lorsque p_j reçoit ce message, si $ts_j = ts'$, alors $cles_j[j] = c$, car cette variable ne peut pas avoir changé de valeur depuis que p_j a envoyé le premier Reponse ($cles_j[j]$ ne peut changer de valeur que lorsque ts' prend une nouvelle valeur, plus grande). Donc, comme on a alors $ts_i \geq tsl_i > ts'$, cela signifie que que ts_j va prendre pour valeur $ts_i > ts'$, ligne 36. Et ensuite, le message Reponse envoyé à p_i va contenir un $ts'' = ts_j > ts'$, validant ce lemme. Sinon, à réception de ce second message Requete, $ts_j \neq ts'$, ce qui signifie (comme ts' était contenu dans ts_j et que les valeurs successives de ts_j sont croissantes) que $ts_j > ts'$. Donc, encore une fois, le message Reponse envoyé contiendra $ts'' > ts'$. □

Lemme 4.3.24. *Soit p_i un processus correct. Chaque appel de l'opération Lecture() par p_i retourne.*

Démonstration. Supposons par contradiction qu'une opération de Lecture() L par p_i ne termine pas. D'après l'algorithme, cela signifie que p_i attend pour toujours à la ligne 18, car le nombre d'itérations ne peut pas dépasser N et donc la boucle extérieure ne peut pas être infinie. Donc, à partir d'un certain temps, on a $|Q_i \cup Ql_i| < n - f$ pour toujours.

Soient $\langle vl, tsl \rangle$ et s les valeurs respectivement stockées dans $\langle vl_i, tsl_i \rangle$ et seq_i immédiatement avant que p_i n'arrive à la ligne 18 de son attente infinie. On peut noter que les variables $\langle vl_i, tsl_i \rangle$ et seq_i ne sont plus modifiées une fois que p_i a commencé à attendre, et que Q_i et Ql_i ne sont plus réinitialisés à $\emptyset$ à partir de ce moment.

Soit p_j un processus correct. Comme p_i est correct et que $\langle vl, tsl \rangle$ est initialement stocké dans $\langle v_i, ts_i \rangle$, alors tout message de Requete envoyé par p_i est de la forme (Requete, $seq, \langle v, ts \rangle, *, *, *, *$) avec $ts \geq tsl = tsl_i$ et $seq = seq_i$. En particulier, à chaque fois que p_i envoie un tel message à p_j, d'après le lemme 4.3.23, p_i reçoit un message Reponse contenant $\langle v', ts' \rangle$. De plus, si $ts' < tsl$, alors p_i envoie un nouveau message Requete, et reçoit ensuite un autre message Reponse avec $\langle v'', ts'' \rangle$ et $ts'' > ts'$. Par récurrence, et comme il n'existe qu'un nombre fini de valeurs possibles inférieures à tsl pour l'estampille contenue dans ces messages, au bout d'un certain nombre d'échanges, p_i reçoit un message Reponse avec $ts \geq tsl$. Lorsque p_i reçoit un tel message, comme le numéro de séquence seq_i est toujours inchangé, il ajoute p_j à Q_i ou à Ql_i (lignes 28 et 29).

Donc, au bout d'un certain temps, tous les processus corrects apparaissent dans $Q_i \cup Ql_i$, ce qui signifie que $|Q_i \cup Ql_i| \geq n - f$. Donc l'attente de p_i ligne 18 se termine, et donc l'opération de Lecture() aussi. □

Lemme 4.3.25. *Supposons que l'écrivain p_n soit un processus correct. Alors, chaque appel à l'opération Ecriture() par p_n termine.*

Démonstration. Supposons par contradiction qu'une opération d'Ecriture() E ne termine pas. Soit v le paramètre de cette opération, et ts l'estampille associée (ligne 7). Soit s la valeur stockée dans seq_n après la ligne 7. Soit p_i un processus correct. Au début de E, $\langle v_n, ts_n\rangle$ prend pour valeur $\langle v, ts\rangle$. Comme p_n est supposé correct, et comme pour la démonstration du lemme précédent, au bout d'un certain temps tout processus correct est dans $Q_n \cup Ql_n$. De plus, comme les nouvelles estampilles ne sont introduites que lors d'opération d'Ecriture(), on peut déduire du fait que E ne termine pas que ts est la plus grande estampille (car la plus récente). Donc, il n'est pas possible qu'un processus correct p_i soit dans Ql_n, car cela ne peut arriver que si son estampille ts_i envoyée dans un message de Reponse est strictement plus grande que ts. Cela signifie alors que tous les processus corrects sont dans Q_i, d'où $|Q_i| \geq n - f$. Donc l'opération E termine, ce qui démontre ce lemme. □

Propriété 4.3.26. *Cet algorithme vérifie la propriété de Terminaison des α-registres.*

Le lemme suivant est utile dans la démonstration des propriétés de Cohérence Finale et de Propagation.

Lemme 4.3.27. *Si un processus correct p_i a $\langle v_0, ts_0\rangle$ dans son $\langle v_i, ts_i\rangle$ à un instant t_0, et si une infinité d'opérations de Lecture sont exécutées, alors à partir d'un instant t_1, chaque processus correct p_j vérifie $\langle v_j, ts_j\rangle \succeq \langle v_0, ts_0\rangle$.*

Démonstration. Supposons par contradiction qu'un processus correct p_j vérifie pour toujours $\langle v_j, ts_j\rangle \prec \langle v_0, ts_0\rangle$. Soit p_k un processus correct qui appelle Lecture() une infinité de fois. En particulier, p_k envoie une infinité de messages de type Requete à p_i, et p_i reçoit chacun de ces messages après un certain temps, et y répond par un message de type (Reponse, $*, \langle v, ts\rangle, *, *, c$) avec $\langle v, ts\rangle \succeq \langle v_0, ts_0\rangle$, qui sont eux aussi reçus au bout d'un certain temps.

Supposons dans un premier temps que p_k vérifie lui aussi pour toujours $\langle v_k, ts_k\rangle \prec \langle v_0, ts_0\rangle$. Lorsqu'un message de type Reponse de p_i est reçu par p_k, si $c = cles_k[k]$, alors p_k doit mettre sa valeur à jour, ce qui est impossible par hypothèse (ligne 26). Donc, chacun de ces messages de réponse vérifie $c \neq cles_k[k]$. Or, ce c est la valeur qui était notée *sa_cle* dans le message de type Requete envoyé par p_k (et auquel correspond la réponse), et qui était donc stockée dans $cles_k[k]$ au moment de l'envoi de cette requête. Cela signifie qu'entre le moment où p_k envoie un message de Requete à p_i, et le moment où le message de Reponse correspondant est reçu, $cles_k[k]$ a changé de valeur. Comme ces échanges arrivent une infinité de fois (car p_k exécute une infinité de Lecture), $cles_k[k]$ est modifié une infinité de fois. De plus, à chaque fois que $cles_k[k]$ est modifié, $\langle v_k, ts_k\rangle$ prend une nouvelle valeur, qui est nécessairement plus grande. Or, il n'existe qu'un nombre fini de couple $\langle v, ts\rangle$ plus petits que $\langle v_0, ts_0\rangle$. Donc $\langle v_k, ts_k\rangle$ finit par prendre une valeur $\langle v_1, ts_1\rangle \succeq \langle v_0, ts_0\rangle$.

Soit t_2 un instant après lequel $\langle v_k, ts_k\rangle \succeq \langle v_0, ts_0\rangle$. Comme p_k effectue une infinité de Lecture(), il envoi une infinité de message de Requete à p_j, chacun de ces messages contenant un $\langle v, ts\rangle \succeq \langle v_k, ts_k\rangle \succeq \langle v_0, ts_0\rangle$. De son côté, p_j répond par des messages de type Reponse contenant la valeur actuelle de $cles_j[j]$. De même que précédemment, si p_j ne modifie pas son $\langle v_j, ts_j\rangle$ après réception de ces requêtes, c'est que $cles_j[j]$ a changé entre le moment où p_j a envoyé une Reponse à p_k, et le moment où p_j reçoit une Requete qui a été envoyée par p_k après réception de cette Reponse (ce qui arrive car p_k appelle Lecture() infiniment souvent). Donc, encore une fois, cela signifie que $cles_j[j]$ change de valeur infiniment souvent, et donc que $\langle v_j, ts_j\rangle$ aussi. D'où on déduit qu'après un certain temps, $\langle v_j, ts_j\rangle \succeq \langle v_0, ts_0\rangle$. □

Ce lemme signifie que la propriété de propagation est vérifiée, car tout argument de Ecriture ou valeur de retour de Lecture est contenu dans la variable v_i du processus en

question. De plus, la propriété de cohérence finale est un corollaire de la propagation, d'où :

Propriété 4.3.28. *Cet algorithme vérifie les propriétés de Cohérence Finale et de Propagation des α-registres.*

Il reste à démontrer que $|V_v| \leq 2M - 1$, et pour cela on utilise les mêmes notations que pour l'algorithme précédent, à savoir V_E, V_L, V_v, V_p, V_m, U, présentées dans la figure 4.6. En particulier, le lemme 4.3.12 et le corollaire 4.3.13 sont toujours vrais pour cet algorithme, d'où $|V_p| \leq f + 1$.

Lemme 4.3.29. *$|U| \leq f$, où $U = \{v \in V_m : \exists \tau \in \mathcal{I}$ et $\exists p_i$ tels que $v_i^\tau = v\}$.*

Démonstration. Soit $u \in V_m$ et ts son estampille. Supposons qu'à un certain instant $\tau \in \mathcal{I}$, $\langle v_x^\tau, ts_x^\tau \rangle = \langle u, ts \rangle$ pour un processus p_x. Au début de $\mathcal{I}$, aucun processus ne stocke localement u ($\forall p_j, v_j^{\tau_0} \neq u$), mais un message contenant u a été envoyé et pas encore reçu. Soit p_i le premier processus qui modifie sa paire $\langle v_i, ts_i \rangle$ pour prendre pour valeur $\langle u, ts \rangle$ (ligne 26 ou 36). Comme $u \notin V_p$, cela arrive lorsque p_i reçoit un message $m = (*, *, \langle u, ts \rangle, *, *, *)$ envoyé à p_i avant $\mathcal{I}$. Et donc il existe un processus p_j tel que $m \in Mess_{j \to i}$.

Considérons une autre valeur $u' \neq u$ qui, de la même manière que u, (1) est dans V_m et (2) à un instant $\tau' \in \mathcal{I}$ vérifie $v_{x'}^{\tau'} = u'$ pour un processus $p_{x'}$. Soit $p_{i'}$ le premier processus qui modifie son $v_{i'}$ en prenant u' pour valeur. Cela arrive à la réception d'un message $m' \in Mess_{j' \to i'}$.

Supposons par contradiction que $p_i = p_{i'}$. Sans perdre en généralité, on supposera que p_i change sa valeur v_i pour u en premier, puis plus tard pour u'. D'après l'algorithme, lorsque $\langle v_i, ts_i \rangle$ est modifié, $cles_i[i]$ est incrémenté. De plus, comme les valeurs successives de $cles_i[i]$ sont croissantes, et que les messages de $Mess_{j \to i}$ contiennent (dans le champ *vieille_cle*) une valeur qui était stockée dans $cles_i[i]$ un certain temps avant cet envoi, cela signifie qu'après cela, les messages en question contiennent une valeur $vc < cles_i[i]$. Donc, après réception d'un tel message, p_i ne peut pas modifier son $\langle v_i, ts_i \rangle$ (ligne 24 ou 34), en particulier pour prendre la valeur $\langle u', ts' \rangle$, ce qui est une contradiction.

Finalement, on peut noter que ni p_i ni $p_{i'}$ ne peuvent être contenus dans Q_d, puisque pour chaque processus $p_y \in Q_d$, v_y ne peut être remplacé que par une valeur de V_E (lemme 4.3.12). Comme $|Q_v| \geq n - f$ et que chaque valeur $u \in U$ doit être initialement reçue par un processus différent pour être acceptée, on peut conclure que $|U| \leq f$. □

Lemme 4.3.30. *Soit $L \in \mathcal{L}$ une opération de Lecture() et v sa valeur de retour. Alors, $v \in V_E$ ou il existe un ensemble Q_L d'au moins $n - f$ processus tel que $\forall p_i \in Q_L$, il existe un instant τ pendant l'exécution de L tel que $v_i^\tau = v$.*

Démonstration. Supposons par contradiction que ce lemme est faux, et donc en particulier $v \notin V_E$. Soient ts l'estampille associée à v, et p_i le processus qui a appelé l'opération L. La boucle **répéter** (lignes 14 à 20) exécutée par p_i termine après N itérations ou quand $|Q_i| \geq n - f$. Dans le second cas, cela signifie que p_i reçoit un message de Reponse contenant $\langle v, ts \rangle$ de chacun des processus $p_j \in Q_i$ (ligne 29). Donc, pour chacun de ces processus, il y a un moment durant l'exécution de L pour lequel $\langle v_j, ts_j \rangle = \langle v, ts \rangle$, car chaque message reçu qui modifie Q_i a été envoyé après le début de L, comme le garantie le numéro de séquence seq_i (ligne 27) et le système de Requete-Reponse. Comme on suppose que le lemme est faux, ce cas ne peut pas arriver.

Donc, p_i exécute N itérations de la boucle **répéter**. Dans chaque itération associée à un numéro de séquence $seq_i = s$, p_i doit recevoir au moins un message de la forme (Reponse, $*, \langle u^s, ts^s \rangle, s, *, vc$) avec $\langle v_i^s, ts_i^s \rangle \prec \langle u^s, ts^s \rangle$, où $\langle v_i^s, ts_i^s \rangle$ est la valeur de $\langle v_i, ts_i \rangle$ au début de cette itération (c'est-à-dire la valeur stockée dans $\langle vl_i, tsl_i \rangle$ pour cette itération). En

effet, pour terminer cette itération sans avoir $|Q_i| \geq n - f$, p_i doit vérifier $|Ql_i| \geq 1$, et donc recevoir un message avec une valeur strictement plus récente que $\langle v_i^s, ts_i^s \rangle$.

Or, le vc contenu dans le message de Reponse en question était contenu dans $cles_i[i]$ au début de cette itération (et envoyé avec le message de Requete), car cette réponse est garantie de correspondre à la bonne requête grâce au numéro de séquence s. Donc, lorsque p_i reçoit un tel message de p_j, si $cles_i[i]$ n'a pas changé, alors $\langle v_i, ts_i \rangle$ est modifié pour prendre pour valeur $\langle u^s, ts^s \rangle$ (ligne 26). À l'inverse, si $cles_i[i]$ a été modifié, cela signifie que $\langle v_i, ts_i \rangle$ a été modifié aussi, et en particulier contient une valeur $\succ \langle v_i^s, ts_i^s \rangle$. Ainsi, dans tous les cas, à chaque itération de la boucle **répéter**, $\langle v_i, ts_i \rangle$ prend une nouvelle valeur, et on peut noter $\langle v_i^{s+1}, ts_i^{s+1} \rangle \succ \langle v_i^s, ts_i^s \rangle$.

Maintenant, on sait que l'opération de Lecture() est incluse dans l'intervalle $\mathcal{I}$ où toute nouvelle valeur écrite appartient à V_E. Les valeurs possibles que peut prendre $\langle v_i, ts_i \rangle$ au cours de cette opération (y compris initialement) sont dans $V_p \cup U \cup V_E$, par définition de ces ensembles. Or, d'après le corollaire 4.3.13 et le lemme 4.3.29, $|V_p \cup U| \leq 2f+1 = N-1$. On sait aussi que chaque valeur de V_E est strictement plus récente que chaque valeur de $V_p \cup U$. Comme les valeurs successives de $\langle v_i, ts_i \rangle$ sont croissantes (observation 4.3.20), on en déduit qu'en $N-1$ itérations, v_i a changé de valeur au moins $N-1$ fois, et a donc atteint une valeur dans V_E. Donc, après une itération supplémentaire, qui arrive nécessairement, la valeur de vl_i est dans V_E, et donc la Lecture retourne une valeur de V_E. Cela contredit l'hypothèse que $v \notin V_E$, et prouve ce lemme. □

Lemme 4.3.31. $|V_p \cap V_L| \leq M = 2f - n + 2$

Démonstration. La démonstration est identique à celle du lemme 4.3.17, grâce au lemme 4.3.30. □

Lemme 4.3.32. $|V_m \cap V_L| \leq M - 1 = 2f - n + 1$

Démonstration. Comme dans la preuve du lemme 4.3.18, soient $u_1, \ldots, u_x$ les valeurs de $V_m \cap V_L$, ordonnées de la plus ancienne à la plus récente. Comme la valeur u_1 est retournée par une opération de Lecture() et que $u_1 \notin V_E$, il existe un ensemble Q_1 de $n-f$ processus qui ont $v_j = u_1$ à un moment durant cette opération et donc durant $\mathcal{I}$.

Soit $u' \in V_m \cap V_L$, $u' \neq u_1$ (si un tel u' n'existe pas, le lemme est vrai), associé à l'estampille t'. On démontre que le premier processus p_i qui modifie son v_i pour prendre pour valeur u' n'appartient pas à Q_1. Supposons par contradiction que $p_i \in Q_1$, et donc que $Mess_{k \to i}$ contienne un message $(*, *, \langle u', t' \rangle, *, *, vc)$. Comme $\langle u_1, t_1 \rangle \notin V_p$, $\langle v_i, ts_i \rangle$ doit être modifié pour contenir $\langle u_1, t_1 \rangle$ à un moment dans $\mathcal{I}$. Quand cela arrive, $cles_i[i]$ est modifié (ligne 26 ou 36). Par l'observation 4.3.20, $\langle v_i, ts_i \rangle = \langle u', t' \rangle$ ne peut arriver qu'après que $\langle v_i, ts_i \rangle = \langle u_1, t_1 \rangle$, car $t' > t_1$, donc seulement après que $cles_i[i]$ ait été modifié. Comme les valeurs successives de $cles_i[i]$ sont croissantes, et que le vc contenu dans le message qui contient $\langle u', t' \rangle$ était stocké dans $cles_i[i]$ un certain temps avant l'envoi de ce message et donc avant $\mathcal{I}$, lorsque ce message est reçu, $vc < cles_i[i]$. Donc ce message ne peut pas modifier la valeur de $\langle v_i, ts_i \rangle$, ce qui est une contradiction.

Donc $p_i \notin Q_1$. On peut aussi noter que $p_i \notin Q_d$ et que $Q_1 \cap Q_d = \emptyset$, car les processus de Q_d ne peuvent stocker dans v_j que des valeurs plus récentes ou égales à v_d et que $v_d \succ u_1$ (lemme 4.3.12). Donc, pour chaque valeur $u_j \in (V_m \cap V_L) \setminus \{u_1\}$, le premier processus durant $\mathcal{I}$ tel que $v_i = u_j$ n'appartient pas à $Q_1 \cup Q_d$. Enfin, comme lors de la preuve du lemme 4.3.29, pour tout $u_i \neq u_j \in V_m \cap V_L$, les premiers processus à obtenir $v = u_i$ et $v = u_j$ sont distincts. Donc, $|V_m \cap V_L| \leq n - |Q_1| + 1 - |Q_d| \leq 2f - n + 1$. □

Toute valeur retournée lors d'une Lecture() contenue dans l'intervalle $\mathcal{I}$ appartient à $V_p \cup V_m \cup V_E$. Comme $|V_p \cap V_L| \leq 2f - n + 2$ (lemme 4.3.31), et $|V_m \cap V_L| \leq 2f - n + 1$ (lemme 4.3.32), on a $|V_v| = |V_L \cap (V_p \cup V_m)| \leq 2f - n + 2 + 2f - n + 1 = 2M - 1$. Cela prouve le lemme :

Lemme 4.3.33. *$|V_v| \leq 2M - 1$ où $M = max(1, 2f - n + 2)$.*

Finalement, on peut déduire que :

Théorème 4.3.34. *L'algorithme 4.4 implémente un α-registre avec $\alpha = 2M - 1$.*

4.4 Borne inférieure sur les possibilités de simulation d'α-registres

Grâce aux algorithmes précédents, on sait qu'il est possible d'implémenter des α-registres avec $\alpha \geq 2M - 1$, mais est-il possible de faire mieux ? On développe ici une borne inférieure, comme dans [29], en montrant que, pour tout algorithme déterministe qui implémente un α-registre pour un certain α, alors on a nécessairement $\alpha \geq M$. On améliore ensuite cette borne en démontrant que $\alpha \geq M + 1$.

4.4.1 Première borne

Pour cette démonstration, on considère une exécution particulière dans laquelle aucune écriture n'a lieu durant l'intervalle $\mathcal{I}$, et donc toute valeur lue est *vieille*, c'est-à-dire $V_L = V_v$. De plus, on construit une famille d'exécutions pour lesquelles aucun processus ne tombe en panne (tous les processus sont corrects). Ces exécutions sont représentées schématiquement sur la figure 4.8.

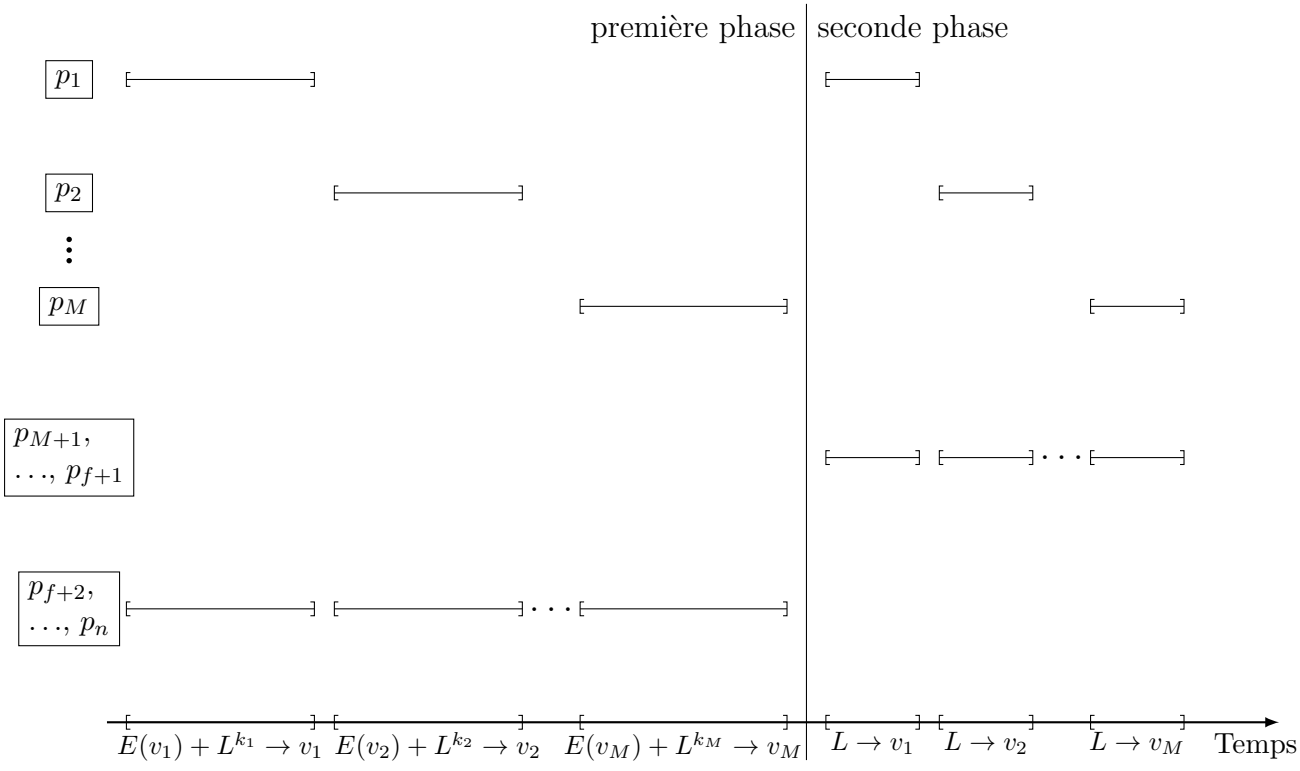

FIGURE 4.8 – Exécution pour la démonstration de la borne inférieure. Chaque intervalle successif représente une ronde de l'exécution, et les processus qui participent (activement) à cette ronde

Théorème 4.4.1. *Soit un système distribué asynchrone par envoi de messages à pannes, tel que $f \geq \frac{n}{2}$. Pour tout algorithme implémentant un α-registre dans ce modèle, $\alpha \geq M = 2f - n + 2$.*

Démonstration. Soit A un algorithme qui implémente un α-registre, où p_n est le processus écrivain. Sans perdre en généralités, supposons que cet algorithme soit à information complète, c'est-à-dire que l'état de chaque processus contient l'historique complet de ce processus, et que chaque message envoyé par un processus contient son état.

On construit une famille d'exécutions de A, chacune paramétrée par M entiers naturels $k_1, \ldots, k_M$. On montre que, pour certaines valeurs de $k_1, \ldots, k_M$, M valeurs distinctes sont retournées par des opérations de Lecture durant un intervalle de temps où aucune Ecriture n'est en cours. Chaque exécution est divisée en deux phases, et chacune de ces phases est constituée de M rondes séquentielles.

Soit $V = \{v_1, \ldots, v_M\}$ un ensemble de M valeurs distinctes. Pour chaque $i \in [1, M]$, soit Q_i et Q'_i les ensembles de $n - f$ processus $Q_i = \{p_i\} \cup \{p_{f+2}, \ldots, p_n\}$ et $Q'_i = \{p_i\} \cup \{p_{M+1}, \ldots, p_{M+n-f-1}\}$. On observe que $Q_i \cap Q'_i = \{p_i\}$ car $M + n - f - 1 = f + 1$. Pour le M-uplet $k = (k_1, \ldots, k_M)$, on définit l'exécution E_k comme suit :

Première phase. Cette phase est constituée de M rondes $r_1, \ldots, r_M$. Chaque ronde r_{i+1} débute après la fin de la ronde r_i. Seuls les processus dans Q_i effectuent des pas de calcul durant la ronde r_i, à cause de l'asynchronisme, et les messages envoyés ne sont pas reçu avant la fin de la seconde phase, sauf mention explicite du contraire.

Au début de la ronde r_i, chaque processus de Q_i reçoit tous les messages qui leur étaient destinés et qui ont été envoyé lors des rondes précédentes. Ensuite, $k_i + 1$ opérations sur l'α-registre implémenté par A sont exécutées de manière séquentielle (c'est-à-dire qu'une opération n'est appelée qu'après que la précédente ait terminé), dans cet ordre :

1. p_n exécute une opération Ecriture(v_i) ;
2. p_i exécute successivement k_i opérations de Lecture().

Au cours de ces opérations, tous les messages envoyés à des processus $p_j \notin Q_i$ sont arbitrairement ralentis et donc ne sont pas reçus, mais les messages destinés aux processus dans Q_i sont correctement reçus rapidement. Comme $|Q_i| = n - f$, cette exécution est indistinguable de celle qui est identique à E_k jusqu'à la fin de la ronde r_{i-1}, puis dans laquelle tous les processus $p_j \notin Q_i$ tombent en panne au début de r_i. Comme l'algorithme A tolère f pannes et vérifie la propriété de Terminaison, chacune des opérations de la ronde r_i termine.

Par la propriété de propagation, on sait que si p_i exécute une infinité d'opération de Lecture et qu'aucune autre Ecriture n'a lieu, au bout d'un certain temps chaque Lecture retournera v_i (car p_n est un processus correct ayant écrit v_i, qui est ainsi la valeur la plus récente). Donc, en choisissant un k_i suffisamment large, au moins une opération de Lecture par p_i retournera v_i.

Seconde phase. Cette phase est aussi découpée en M rondes $r'_1, \ldots, r'_M$, telles que la ronde r'_{i+1} commence après la fin de la ronde r'_i. De plus, r'_1 commence après la fin de r_M, et durant la ronde r'_i, seuls les processus dans Q'_i effectuent des pas de calcul

Au début de la ronde r'_i, les processus de Q'_i reçoivent les messages qui leur ont été envoyés durant les rondes r'_j et r_j avec $j \leq i$, mais pas ceux envoyés durant les rondes $r_{j'}$ avec $j' > i$. Cela ne pose pas de problème vis-à-vis du côté FIFO des canaux de communication, car l'ensemble des processus autorisés à envoyer des messages lors des rondes r'_j, $j \leq i$, a une intersection vide avec ceux qui peuvent envoyer des messages durant les rondes $r_{j'}$, $j' > i$. En effet, les processus effectuant des pas de calcul sont $\{p_{i+1}, \ldots, p_M\} \cup \{p_{f+2}, \ldots, p_n\}$ dans le cas des rondes $r_{j'}$, $j' > i$, et $\{p_1, \ldots, p_i\} \cup \{p_{M+1}, \ldots, p_{f+1}\}$ pour les rondes r'_j, $j \leq i$.

Ensuite, le processus p_i exécute une opération de Lecture, et les messages destinés aux processus $p_j \notin Q'_i$ ou envoyés par ces mêmes $p_j \notin Q'_i$ sont arbitrairement ralentis,

contrairement à ceux entre les processus de Q'_i. Cette exécution est indistinguable pour les processus de Q'_i de celle identique à E_k jusqu'à la ronde r'_{i-1}, à la fin de laquelle les processus $p_j \notin Q'_i$ tombent en panne. Comme l'algorithme A tolère f pannes et vérifie la propriété de Terminaison, cette opération de Lecture par p_i termine.

En supposant que k_i a été choisi suffisamment grand, la Lecture précédente de p_i (lors de la ronde r_i) a retourné v_i. D'après la propriété de Monotonie des Lectures, cette opération dans r'_i doit retourner une valeur v plus récente ou égale à v_i, c'est-à-dire une valeur v_j avec $j \geq i$. Comme aucun des messages envoyés durant les rondes $r_{i+1}, \ldots, r_M$ n'a été reçu par les processus de $\bigcup_{j \leq i} Q'_j$, l'exécution similaire à E_k, mais durant laquelle les rondes $r_{i+1}, \ldots, r_M$ n'ont pas eu lieu, est indistinguable de E_k du point de vue des processus de Q'_i. Donc, comme dans cette exécution, v_i est la valeur la plus récente, la Lecture de la ronde r'_i doit retourner v_i.

Lors de la seconde phase, aucune opération d'Ecriture n'est en cours, et M valeurs distinctes sont retournées par des opérations de Lecture lors de cette période. Donc $\alpha \geq M$. □

4.4.2 Amélioration de cette borne

Il est possible d'améliorer la borne précédente, en ajoutant certains aspects à la démonstration précédente.

Théorème 4.4.2. *Soit un système distribué asynchrone par envoi de messages à pannes, tel que $f \geq \frac{n}{2}$. Pour tout algorithme implémentant un α-registre dans ce modèle, $\alpha \geq M + 1$.*

Démonstration. On considère les mêmes hypothèses que dans la preuve du théorème 4.4.1. De plus, on suppose sans perdre en généralités que chaque message envoyé au cours de l'algorithme A est *broadcasté*, c'est-à-dire envoyé de manière identique à *chaque* processus du système. On développe une exécution $E'_{k'}$ de A, dans laquelle $M + 1$ valeurs distinctes sont lues dans un intervalle de temps sans Ecriture active.

Pour définir $E'_{k'}$, on part de l'exécution E_k définie dans la démonstration du théorème 4.4.1, et on ajoute deux rondes spéciales notées r_0 et r'_0. On note $k' = (k_0, k_1, \ldots k_M)$ le $M + 1$-uplet d'entiers naturels associé à cette exécution, et on étend l'ensemble des valeurs à $V = \{v_0\} \cup \{v_1, \ldots, v_M\}$, qui sont $M + 1$ valeurs distinctes. Comme E_k, $E'_{k'}$ est divisé en deux phases, mais chacune est ici composée de $M + 1$ rondes (au lieu de M) séquentielles. Les rondes r_i et r'_i avec $i \in [1, M]$ sont identiques à celles de E_k décrites dans la démonstration du théorème 4.4.1.

Première phase. La ronde spéciale r_0 prend place entre les rondes r_1 et r_2 (à noter que $M \geq 2$, donc r_2 existe nécessairement), c'est-à-dire que r_0 ne commence qu'après la fin de r_1, et r_2 ne commence qu'après la fin de r_0. Les autres rondes se déroulent normalement, comme précédemment. La ronde spéciale r_0 est découpée en deux parties, qui se déroulent l'une après l'autre.

Lors de la première partie de r_0, chaque message envoyé à un processus $p_j \notin Q_M$ est arbitrairement ralenti, et seuls les processus dans Q_M effectuent des pas de calcul. Au début de cette partie, chaque processus de Q_M reçoit les messages envoyés lors de la ronde r_1. Ensuite, le processus exécute une opération Ecriture(v_0), qui termine malgré l'absence d'échanges de messages avec les processus $\notin Q_M$, par propriété de Terminaison.

Lors de la seconde partie, les processus $p_j \notin Q'_1 \cup \{p_n\}$ n'effectuent aucun pas de calcul et ne reçoivent aucun message. De plus, les processus $p_i \in Q'_1$ ne reçoivent de messages envoyés par un processus $\notin Q'_1$ que lorsque ceci est explicitement spécifié. D'abord, l'ensemble des processus de Q'_1 reçoit les messages envoyés lors de la ronde r_1. Ensuite, la seconde partie est constituée de k_0 répétitions successives de l'ensemble d'étapes suivant :

1. Le processus p_1 exécute une opération de Lecture(), qui termine malgré l'absence de message reçu de la part des processus $\notin Q'_1$;
2. Si au moins un message précédemment envoyé par un $p_j \notin Q'_1$ n'a pas encore été reçu par un processus de Q'_1, alors le plus ancien de ces messages est reçu par chacun des processus de Q'_1, et il s'agit d'un même message car il a été *broadcasté*. Il s'agit soit d'un message envoyé lors de la première partie de r_0, soit d'un message envoyé par p_n lors de la seconde partie de r_0 (auquel cas tous les messages envoyés lors de la première partie de r_0 aux processus de Q'_1 ont été reçus) ;
3. p_n reçoit tous les messages qui lui ont précédemment été envoyé, et peut y répondre en envoyant des messages, mais ces derniers sont arbitrairement ralentis, et ne seront reçu que lorsque spécifié explicitement.

Cette exécution est indistinguable de celle dans laquelle les processus $\notin Q_1 \cup Q'_1 \cup Q_M$ sont en pannes depuis le début, les processus de $Q_1 \setminus (Q'_1 \cup Q_M)$ tombent en panne à la fin de la ronde r_1, et les processus dans $Q_M \setminus \{p_n\}$ (dont aucun n'est dans Q'_1) tombent en panne à la fin de la première partie de r_0. Comme p_n est correct, par propriété de Propagation, si k_0 est suffisamment grand, les opérations de Lecture() de r_0 finissent par retourner v_0 (qui est la valeur la plus récente).

Soit K_0 la valeur du plus petit k_0 tel que la dernière Lecture() exécutée par p_1 lors de r_0 retourne v_0. On sait que $K_0 > 1$, puisque la première Lecture retourne v_1 (si k_1 est suffisamment large, $\perp$ sinon), car aucun message envoyé lors de r_0 n'a encore été reçu par les processus de Q'_1. En particulier, on a $K_0 - 2 \geq 0$. On peut noter que K_0 peut dépendre de la valeur de k_1.

Seconde phase. Cette phase est composée de $M + 1$ rondes, les r'_i avec $i \geq 1$ étant toujours identiques à leur version de la démonstration du théorème 4.4.1. La ronde spéciale r'_0 a lieu entre les rondes r'_1 et r'_2, c'est-à-dire que r'_0 commence après la fin de r'_1, et que r'_2 commence après la fin de r'_0.

Au début de r'_0, chaque processus de Q'_1 reçoit le plus ancien message qui leur a été envoyé lors de r_0 (comme lors d'une itération de la 2ème étape de la seconde partie de r_0), si un tel message existe (ce qui est nécessairement le cas si k_0 est suffisamment faible). Ensuite, à part cet unique message (identique pour chacun des processus de Q'_1), tous les messages sont arbitrairement ralentis sauf ceux d'un processus de Q'_1 à un processus de Q'_1. Le processus p_1 effectue alors une opération de Lecture(), qui termine car $|Q'_1| = n - f$.

Supposons maintenant que $k_0 = K_0 - 2 \geq 0$. Par définition de K_0 et le fait qu'aucun message n'ait été reçu entre la fin de r_0 et le début de r'_1, sans oublier le fait que lors de r'_1 les processus de Q'_1 ne reçoivent des messages que de Q'_1, la Lecture() de r'_1 ne retourne pas v_0, et retourne v_1 si k_1 est suffisamment grand. En effet, il s'agit alors de la $(K_0 - 1)$-ième itération de la Lecture de la seconde partie de r_0, et seule la K_0-ième retourne v_0. Ensuite, au début de r'_0, les processus de Q'_1 reçoivent un unique message, exactement comme lors de la $(K_0 - 1)$-ième itération de la 2ème étape de la seconde partie de r_0. Finalement, l'effet de la 3ème étape d'une telle itération n'est pas visible par les processus de Q'_1 tant qu'ils ne reçoivent pas de message supplémentaire. Enfin, lors de la Lecture de r'_0, la situation pour les processus de Q'_1 est indistinguable de celle dans laquelle la ronde r_0 est toujours en cours, et il s'agit de sa K_0-ième itération. Ainsi, par définition de K_0, la Lecture de r'_0 retourne v_0.

Les rondes r'_i pour $i > 1$ s'exécutent alors exactement comme lors de la démonstration du théorème 4.4.1, retournant les valeurs correspondantes v_i si le k_i associé est suffisamment grand.

Donc, en choisissant des k_i suffisamment grands pour chaque $i > 0$, et en prenant ensuite $k_0 = K_0 - 2$ (qui peut dépendre de k_1), on a bien une exécution $E'_{k'}$ dans laquelle les Lecture de la seconde phase retournent les valeurs $\{v_0, v_1, \ldots, v_M\}$. On a donc un

intervalle de temps dans lequel aucune opération d'Ecriture n'est en cours, et $M+1$ valeurs distinctes sont retournées, ce qui prouve que $\alpha \geq M+1$. □

4.4.3 Et au-delà ?

Nous savons maintenant qu'il est impossible d'implémenter un α-registre avec un α plus petit que $M+1$. De plus, dans la partie précédente, deux algorithmes sont détaillés permettant d'implémenter des α-registres avec $\alpha \geq 2M-1$.

La question est alors : où se situe réellement la limite entre le possible et l'impossible ? Malgré l'écart entre les 2 bornes connues que sont $M+1$ et $2M-1$, on peut néanmoins noter que cet écart n'est qu'un simple facteur multiplicatif de 2, et ce pour toutes les valeurs possibles de n et f.

4.5 Composer des α-registres

4.5.1 Un α-registre multi-écrivains ?

Les propriétés d'un α-registre sont décrites comme s'il était possible d'avoir un α-registre multi-écrivains. On est donc en droit de se demander s'il ne serait pas possible d'implémenter un tel objet. Mais un tel objet ne peut pas être implémenter en présence d'une majorité de pannes.

Lemme 4.5.1. *Un α-registre multi-écrivain (c'est-à-dire sans la dernière propriété) ne peut pas être implémenté si $f \geq n/2$.*

Démonstration. Supposons qu'un tel algorithme existe, et considérons les exécutions suivantes : Exécution $\mathcal{E}_1$, tous les processus sauf $p_1, \ldots, p_{n-f}$ tombent en panne, puis p_1 appelle l'opération Ecriture(v_1), qui termine par propriété de Terminaison. Exécution $\mathcal{E}_2$, tous les processus sauf $p_{n-f+1}, \ldots, p_{2(n-f)}$ tombent en panne, puis p_{n-f+1} appelle l'opération Ecriture(v_2), qui termine par propriété de Terminaison. Exécution $\mathcal{E}_3$, tous les processus sauf $p_1, \ldots, p_{2(n-f)}$ tombent en panne, puis p_1 appelle l'opération Ecriture(v_1) et p_{n-f+1} appelle l'opération Ecriture(v_2). De plus, les communications entre un processus parmi $\{p_1, \ldots, p_{n-f}\}$ et un processus parmi $\{p_{n-f+1}, \ldots, p_{2(n-f)}\}$ sont arbitrairement ralenties, dans un sens comme dans l'autre. Alors, cette exécution est indistinguable de l'exécution $\mathcal{E}_1$ par les processus $p_1, \ldots, p_{n-f}$ et de l'exécution $\mathcal{E}_2$ par les processus $p_{n-f+1}, \ldots, p_{2(n-f)}$, donc ces deux opérations d'écritures terminent. Il est alors possible d'avoir une exécution $\mathcal{E}_4$ similaire à $\mathcal{E}_3$ mais dans laquelle l'opération Ecriture(v_1) précède l'opération Ecriture(v_2). De même, on peut avoir une exécution $\mathcal{E}_5$ similaire à $\mathcal{E}_3$ mais dans laquelle l'opération Ecriture(v_2) précède l'opération Ecriture(v_1). Du point de vue de chacun des processus $p_1, \ldots, p_{2(n-f)}$, les exécutions $\mathcal{E}_4$ et $\mathcal{E}_5$ sont indistinguables. Donc, si $\mathcal{E}_4$ et $\mathcal{E}_5$ sont prolongées de la même manière par une même suite d'événements, les processus se comporteront de manière identiques dans un cas comme dans l'autre. Ainsi, si ces exécutions voient suivre une infinité de Lecture() sans aucune Ecriture() supplémentaire, et sans pannes additionnelles, les différentes Lecture() retourneront les mêmes valeurs dans chaque exécution. Or, d'après la propriété de Propagation, les Lecture de $\mathcal{E}_4$ devraient finir par toutes retourner v_2, alors que celles de $\mathcal{E}_5$ devraient finir par toutes retourner v_1. Cette contradiction souligne l'impossibilité d'un α-registre multi-écrivains. □

Comme un α-registre multi-écrivain est impossible, il est alors naturel de se pencher sur l'utilisation de plusieurs α-registres avec des écrivains différents.

4.5.2 Vecteur d'α-registres

On considère désormais un ensemble de n α-registres, chacun avec un écrivain différent. Un tel ensemble est appelé un *vecteur*, noté V, et on désignera par $V[i]$ l'α-registre dont l'écrivain est p_i.

L'utilisation de vecteurs de registres 1WnR n'est pas nouvelle, utilisée notamment avec l'abstraction du *snapshot* [3, 32]. On dispose alors classiquement d'une opération *Collecte* qui permet de lire dans l'ensemble des registres du vecteur en une seule opération. Contrairement au snapshot, l'opération de Collecte ne vérifie pas la linéarisabilité, mais vérifie néanmoins les mêmes propriétés de cohérence qu'un registre régulier. Ainsi, il est possible (même avec des registres atomiques) que deux opérations Ecriture(v_1) sur $V[1]$ et Ecriture(v_2) sur $V[2]$ se déroulent l'une après l'autre, mais que la Collecte concurrente des deux lise la valeur v_2 dans $V[2]$ mais une valeur plus ancienne que v_1 dans $V[1]$. Cependant, avec des registres atomiques, toute valeur lue dans $V[i]$ au cours de cette Collecte est soit la dernière valeur écrite dans $V[i]$ avant cette opération, soit une valeur écrite concurremment.

L'opération en question peut s'écrire comme suit :

1: **Fonction** COLLECTE
2: $Res \leftarrow$ tableau de n valeurs ;
3: **pour** i **de** 1 **à** n **faire**
4: $Res[i] \leftarrow$ LECTURE( ) sur le registre $V[i]$;
5: **retourner** Res ;

Cette algorithme fonctionne de la même manière quel que soit le type de registres utilisés. Cependant, les propriétés que vérifient les valeurs de retour de *Collecte* sont semblables à celles vérifiées par les registres qui compose ce vecteur. Ainsi, *Collecte* utilisé sur un vecteur d'α-registres retournera un tableau de valeurs qui suivent la propriété de Lectures α-bornées. Par exemple, si *Collecte* est utilisé plusieurs fois en absence d'Ecriture, chaque case du tableau de retour contient jusqu'à α valeurs différentes lors des différents appels. Cependant, une propriété comme la cohérence causale sur un tel ensemble d'objets n'est pas nécessairement vérifiée, et peut demander une implémentation attentive, bien qu'elle reste vraie pour tout registre du vecteur considéré individuellement.

4.6 Remarques avant le chapitre suivant

Au cours de ce chapitre, deux objets distincts et indépendants (et même utilisant des concepts très différents) ont été présentés et étudiés. De manière surprenante, la valeur de $2f-n+2$ apparaît en lien avec chacun de ces objets. Plus précisément, il s'agit exactement de la borne inférieure pour une fonction de χ-coloration de quorums, qui est utilisée pour implémenter des bancs de registres χ-colorés. De plus, cette valeur est présente dans la borne inférieure d'implémentation des α-registres, qui est de $(2f - n + 2) + 1$. Enfin, elle apparaît aussi dans la valeur de α pour les algorithmes présentés en partie 4.3, à savoir $2 * (2f - n + 2) - 1$.

Un tel résultat est intéressant à souligner, car non prévu. En effet, ce $2f - n + 2$ survient de deux façons différentes : Dans le premier cas, il s'agit du nombre chromatique du graphe de Kneser $KG_{n,n-f}$, résultat déjà connu [40]. Dans le second, il apparaît pour des exécutions particulières dans lesquelles ont isole 2 ensemble de $n - f - 1$ processus (des *quorums moins 1*), et on considère les $2f - n + 2$ processus restants.

Dans le chapitre suivant cependant, ce $2f-n+2$ n'est pas présent, et une autre borne apparaît : $\lfloor n/(n - f)\rfloor$. On peut remarquer que, lorsque $f = n/2$ (c'est-à-dire lorsqu'on passe juste la barrière de la majorité de pannes), on a $2f - n + 2 = \lfloor n/(n - f)\rfloor = 2$.

De plus, lors du cas extrême $f = n - 1$ (tous les processus sauf 1 peuvent tomber en panne), ce qui correspond notamment au modèle *sans-attente* (*wait-free* en anglais) [], $2f - n + 2 = \lfloor n/(n - f) \rfloor = n$. Enfin, lors des autres cas qui nous intéressent, c'est-à-dire $n/2 < f < n-1$ (le cas $f < n/2$ étant déjà très largement étudié), on peut remarquer que $\lfloor n/(n - f) \rfloor < 2f - n + 2$. Ainsi, ce $\lfloor n/(n - f) \rfloor$ peut donc être considéré comme *meilleur* que $2f - n + 2$, car on cherche à réduire ces bornes autant que possible, et qu'on a toujours $\lfloor n/(n - f) \rfloor \leq 2f - n + 2$, avec une différence entre les deux dans les cas non-extrêmes.

Chapitre 5

Extension d'autres objets partagés : les diviseurs et le renommage

Ce chapitre discute des objets partagés appelés *diviseurs* et du problème du *renommage*, ainsi que de leurs extensions. On présente tout d'abord leurs versions classiques, qui ont été évoquées dans la partie 3.3.2. Ensuite, les diviseurs à-un-coup et les diviseurs réutilisables sont étendus de manière à être implémentables en présence d'une majorité de pannes. Enfin, le renommage est lui aussi étendu, et notamment résolu en utilisant les k-diviseurs précédemment définis.

5.1 Les diviseurs habituels et leur application au renommage

5.1.1 Renommage classique

Le problème du renommage, évoqué dans la partie 1.3.3, a été introduit dans [17]. Il existe en deux versions *à-un-coup* et *réutilisable*. La seconde, ayant été proposée dans [78], sera présentée en partie 5.1.5, mais justifie l'appelation *à-un-coup* de la version du renommage qui fut la première considérée.

Au cours de ce problème, les processus peuvent chercher à obtenir un *nom* x_i, qui pourra par exemple leur servir à modifier leur identité id_i. Ces valeurs (ou noms) ainsi obtenues doivent être uniques, et on cherche qualitativement à ce qu'ils soient contenus dans un intervalle de taille réduite.

Ce problème a été largement étudié [37, 53, 6, 19, 61, 36], dans divers modèles de l'algorithmique distribuée. Il a aussi été analysé en lien avec d'autres problèmes [52, 51]. Une utilité notable de ce problème est liée aux problèmes d'*allocation de ressources* et de *sections critiques* [25, 71, 88, 67]. En effet, ces problèmes représentent notamment des situations dans lesquels plusieurs machines tentent d'accéder simultanément à des ressources partagées et similaires, mais telles que seule une machine peut utiliser une ressource donnée à la fois, et les ressources doivent donc être réparties entre les machines. Ainsi, en associant un nom unique à chaque ressource, les processus représentant les machines cherchent à obtenir un nom unique leur garantissant l'accès à la ressource correspondante. Un exemple plus concret serait alors le partage de plusieurs imprimantes dans un petit réseau informatique : chaque utilisateur qui souhaite imprimer un document ne s'intéresse pas forcément à l'imprimante effectivement utilisée, tant que son document est imprimé, et assigner toutes les tâches à une seule imprimante crée une surcharge de travail et un ralentissement global des impressions.

On donne maintenant une définition plus formelle du problème du renommage. Le

problème du renommage *à-un-coup* a des contraintes d'utilisation particulières :

1. Initialement, chaque processus p_i a un identifiant unique id_i.
2. Chaque processus ne peut participer qu'une seule fois au renommage.

Et des propriétés qu'il doit vérifier si correctement utilisé :

1. Chaque processus correct p_i qui participe au renommage doit retourner une (unique) valeur x_i en un temps fini
2. Si deux processus distincts p_i et p_j retournent deux valeurs x_i et x_j, alors $x_i \neq x_j$.

On considérera que l'ensemble des identifiants initiaux ainsi que celui des valeurs de retour possibles sont contenus dans $\mathbb{N}$. De plus, on pourra restreindre ces ensembles à deux intervalles $[0, I_m]$ et $[0, X_m]$.

On parle de *renommage dans un espace de noms de taille M*, aussi noté M-renommage, un problème de renommage pour lequel l'ensemble des valeurs de retour possibles est de taille M, c'est-à-dire $[0, M-1]$. Intuitivement, lors de l'écriture d'algorithmes pouvant résoudre le problème du renommage, on cherchera à écrire des algorithmes de M-renommage avec un M aussi petit que possible, avec malgré tout de potentiels compromis pour gagner en complexité en temps, complexité en espace, et en simplicité de l'algorithme (voir par exemple [18, 79]).

L'algorithme 5.1 est un exemple d'algorithme de M-renommage qui fonctionne dans notre modèle par envoi de message à condition que $f < n/2$. Il s'agit d'une version reformulée de l'algorithme présent dans [17], et est présenté ici pour deux raisons. La première raison est qu'il permet de mieux comprendre les rouages et enjeux du problème du renommage à-un-coup, notamment la limite à une minorité de pannes qui semble se dessiner. La seconde raison est l'étude de ce même algorithme en présence d'une majorité de pannes, dans la partie 5.2.4.

Cet algorithme fonctionne approximativement comme suit, répété à l'infini jusqu'à ce que tous les processus retournent une valeur ou tombent en panne :

1. Chaque processus p_i communique avec les autres, jusqu'à avoir des données suffisamment à jour et *stables* (lignes 10 – 20)
2. Si p_i a proposé une valeur x, et que d'après ses données, il est le seul processus à avoir fait cela, alors p_i termine son renommage en retournant cette valeur x (ligne 21).
3. Sinon, (p_i n'a rien proposé ou il n'est pas seul à avoir proposé cette même valeur), p_i calcule son *rang* parmi les processus qu'il pense indécis (d'après ses données), c'est-à-dire qui n'ont pas encore retourné une valeur (ligne 27).
4. Si son rang est suffisamment faible, p_i propose une (nouvelle) valeur qui dépend de son rang et de l'ensemble des valeurs proposées par aucun processus (d'après les connaissances de p_i) (ligne 29).

Le *rang* dépend des identifiants (initiaux) des processus, et permet de limiter les risques que plusieurs processus proposent la même valeur, afin de garantir que des processus retournent la valeur qu'ils ont proposée. De plus, limiter la proposition aux processus de rang $\leq f+1$ permet de réduire l'espace de nom de cet algorithme à $n+f$.

Algorithme 5.1 Renommage à-un-coup, par envoi de messages. (code exécuté par p_i)

1: **Initialisation**
2: $V_i \leftarrow$ tableau dynamique de triplets $\langle x, ts, b\rangle$, indexé par des identifiants id, et dont les cases non initialisées valent $\perp$;
3: $Connus_i \leftarrow \{id_i\}$; ▷ indique les id des cases non-$\perp$

```
4:    V_i[id_i] ← ⟨−1, 0, 0⟩ ;
5:    c_i ← 1 ;
6:  Fonction RENOMME(id_i)
7:    pour chaque processus p ≠ p_i du système faire
8:      envoyer le message (Connus_i, V_i) à p ;
9:    tant que vrai faire
10:     tant que c_i < n − f faire
11:       attendre de recevoir un message (C, V) du processus p_j ;
12:       Connus_i ← Connus_i ∪ C ;
13:       si V_i = V alors
14:         c_i ← c_i + 1 ;
15:       sinon si V ⋠ V_i alors
16:         V_i ← FUSIONNE(Connus_i, V_i, V) ;
17:         c_i ← 1 ;
18:         pour chaque processus p ≠ p_i du système faire
19:           envoyer le message (Connus_i, V_i) à p ;
20:     x_i ← V_i[id_i].x ;
21:     si x_i ≠ −1 et ∀id ∈ Connus_i \ {id_i}, V_i[id].x ≠ x_i alors
22:       V_i[id_i].b ← 1 ;
23:       V_i[id_i].ts ← V_i[id_i].ts + 1 ;
24:       Invoque en parallèle la fonction FIN
25:       retourner x_i ;
26:     sinon
27:       r ← RANG_INDECIS(Connus_i, V_i, id_i) ;
28:       si r ≤ f + 1 alors
29:         V_i[id_i].x ← VALEUR_LIBRE(Connus_i, V_i, r) ;
30:         V_i[id_i].ts ← V_i[id_i].ts + 1 ;
31:         c_i ← 1 ;
32:         pour chaque processus p ≠ p_i du système faire
33:           envoyer le message (Connus_i, V_i) à p ;

34: Fonction FUSIONNE(C, V, V′)
35:   pour chaque id ∈ C tel que V′[id] ≠⊥ faire
36:     si V[id] =⊥ ou V[id].ts < V′[id].ts alors          ▷ V[id] < V′[id]
37:       V[id] ← V′[id] ;
38:   retourner V ;
39: Fonction RANG_INDECIS(C, V, id_i)
40:   indecis ← ∅ ;
41:   pour chaque id ∈ C faire
42:     si V[id].b = 0 alors
43:       indecis ← indecis ∪ {id} ;
44:   rang ← 0 ;
45:   pour chaque id ∈ indecis faire
46:     si id ≤ id_i alors
47:       rang ← rang + 1 ;
48:   retourner rang ;
49: Fonction VALEUR_LIBRE(C, V, r)
50:   libre ← tableau de n + f booléens initialisés à vrai (indexé de 0 à n + f − 1) ;
51:   pour chaque id ∈ C faire
```

52: **si** $V[id].x \geq 0$ **alors**
53: $libre[V[id].x] \leftarrow faux$;
54: $val \leftarrow -1$; $restant \leftarrow r$;
55: **tant que** $restant > 0$ **faire**
56: $val \leftarrow val + 1$;
57: **si** $libre[val]$ **alors**
58: $restant \leftarrow restant - 1$;
59: **retourner** $indice$;

60: **Procédure** FIN
61: **tant que** $vrai$ **faire**
62: **pour chaque** processus $p \neq p_i$ du système **faire**
63: **envoyer** le message $(Connus_i, V_i)$ **à** p ;
64: **attendre** jusqu'à recevoir un message (C, V) de p_j tel que $V \not\leq V_i$;
65: $Connus_i \leftarrow Connus_i \cup C$;
66: $V_i \leftarrow$ FUSIONNE$(Connus_i, V_i, V)$;

On utilise dans cet algorithme des tableaux V contenant des $\perp$ et des triplets $\langle x, ts, b\rangle$. On note alors $V[id].x$ la valeur du champ x du triplet contenu dans $V[id]$, et de même $V[id].ts$ et $V[id].b$ pour les valeurs des champs ts et b. On établit une relation d'ordre sur les triplets en notant $\langle x, ts, b\rangle \leq \langle x', ts', b'\rangle \Leftrightarrow ts \leq ts'$ et $\perp < \langle x, ts, b\rangle$. On peut alors étendre cette relation d'ordre en un relation d'ordre partielle sur les tableaux V : $V \leq V' \Leftrightarrow \forall id, V[id] \leq V'[id]$, autrement dit $\forall id$, $V[id] \neq \perp \Rightarrow (V'[id] \neq \perp$ et $V'[id].ts \geq V[id].ts)$. Ainsi, $V \not\leq V'$ signifie $\exists id : V[id] > V'[id]$.

Au cours de cet algorithme, les processus font évoluer le contenu de leur tableau V_i, et se le transmettent des uns aux autres (en envoyant nécessairement une copie conforme de la valeur de leur variable V_i). On dira alors qu'un tableau V est *stable* au cours d'une exécution de cet algorithme si un processus p_i reçoit cet exact tableau V par message de $n - f - 1$ différents autres processus, tout en ayant $V_i = V$.

De plus, on dira que cet algorithme est *localement correct* si pour toute exécution possible, pour chaque processus p_i, les valeurs successives de sa variable V_i sont totalement ordonnées dans un ordre croissant, c'est-à-dire si $\forall \tau < \tau'$ instants d'une exécution, $V_i^\tau \leq V_i^{\tau'}$, où V_i^τ représente la valeur de V_i à l'instant τ.

On démontre alors le lemme suivant :

Lemme 5.1.1. *Si un algorithme A est localement correct et que $f < n/2$, alors tout ensemble de tableaux V* stables *est totalement ordonné*

Démonstration. Supposons par contradiction que, au cours d'une exécution, deux tableaux V et V' soient stables et soient incomparables ($V \not\leq V'$ et $V' \not\leq V$). Comme V est stable, au moins $n - f - 1$ processus différents ont envoyé ce V dans un message, et donc au moins $n - f$ processus p_j ont vérifié $V_j = V$ à un instant durant cette exécution. De même, au moins $n - f$ processus différents ont vérifié $V_j = V'$ à un instant. Comme $f < n/2$, $2(n - f) > n$ et donc il existe au moins un processus p_i qui a contenu V et V' au cours de cette exécution. Or, A est localement correct, donc V et V' sont totalement ordonnés. □

Pour en revenir à l'algorithme 5.1, la fonction Fin permet de continuer à transmettre les V_i après que p_i ait terminé son renommage en retournant une valeur, afin que les autres processus puissent continuer à progresser. La fonction *Rang_indecis* permet de calculer le *rang* de p_i parmi l'ensemble des processus indécis qu'il connaît, c'est-à-dire quelle est la place de son identifiant id_i dans la liste (croissante) des identifiants des processus (connus

par p_i) qui n'ont pas encore retourné de valeur (d'après les données contenues dans V_i). La fonction $Valeur\_libre$ permet alors de retourner la r-ième valeur de l'ensemble (ordonné) des valeurs qui ne sont actuellement proposées par aucun processus (d'après les données dans V_i). Enfin, la fonction $Fusionne$ permet de mettre à jour V_i en gardant toutes les valeurs les plus récentes pour chaque case, et permet donc de démonter le lemme suivant :

Lemme 5.1.2. *L'algorithme 5.1 est localement correct.*

Démonstration. La valeur contenue dans une variable V_i n'est modifiée que pour prendre la valeur retournée par la fonction Fusionne (prenant notamment V_i comme paramètre). Or, cette fonction, à partir de deux tableaux V et V' en produit un V'' tel que $V'' \geq V$ et $V'' \geq V'$. En effet, pour chaque indice id, soit $V'[id] = \perp$ et dans ce cas $V''[id] = V[id]$, soit $V'[id] \neq \perp$. Dans ce second cas, soit $V[id] = \perp$, auquel cas $V''[id] = V'[id]$, soit $V''[id]$ reçoit la valeur maximale de $(V[id], V'[id])$. Dans tous les cas, Fusionne retourne un tableau $V'' \geq V_i$, ce qui suffit à démontrer ce lemme. □

On déduit des deux derniers lemmes que pour toute exécution de cet algorithme, l'ensemble des tableaux V *stables* est totalement ordonné. Ce résultat permet, à l'aide des lemmes 5.1.3 et 5.1.4 de démontrer le lemme 5.1.5 qui garantit la propriété d'unicité du renommage.

Lemme 5.1.3. *Lorsqu'un processus p_i décide d'une valeur x_i à retourner via la fonction Renomme de l'algorithme précédent (ligne 20 puis 25), sa variable V_i contient un tableau* stable.

Démonstration. En effet, pour sortir de la boucle ligne 10, p_i doit vérifier $c_i \geq n - f$. Or, c_i n'est incrémenté que lorsque p_i reçoit d'un autre processus un message contenant l'exact tableau V qui est actuellement stocké dans V_i. De plus, si p_i modifie la valeur de V_i, alors c_i est réinitialisé à 1. Donc, il ne peut sortir de cette boucle que lorsqu'il a reçu au moins $n-f-1$ messages de différents processus, contenant le tableau V_i : il s'agit donc d'un tableau stable. Enfin, on remarque que, entre la fin de cette boucle et l'exécution de la ligne 20, la valeur de V_i ne change pas, ce qui démontre ce lemme. □

Lemme 5.1.4. *Une fois qu'un processus p_i a retourné une valeur x_i pour la fonction Renomme, alors la valeur de $V[id_i]$ ne peut plus contenir de valeur plus grande que celle de $V_i[id_i]$ au moment du retour de x_i par p_i, et $V_i[id_i].x = x_i$. En particulier, tout tableau $V \geq V_i$ vérifie alors $V[id_i] = V_i[id_i]$.*

Démonstration. Si un processus p_j modifie la valeur de son $V_j[id_i]$, c'est soit dans la fonction Fusionne, appelée après avoir reçu un message contenant la valeur qui sera stockée dans $V_j[id_i]$, soit au cours de la fonction Renomme *si* $p_j = p_i$. Or, une fois que p_i a retourné une valeur x_i, il n'exécute plus la fonction Renomme, ce qui explique qu'aucune nouvelle valeur ne peut être créée pour $V[id_i]$. □

Lemme 5.1.5. *Si deux processus p_i et p_j retournent deux valeurs x_i et x_j via la fonction Renomme de l'algorithme 5.1, alors $x_i \neq x_j$.*

Démonstration. Supposons que $x_i = x_j$. Soit V (respectivement V') le tableau stable contenu dans V_i (respectivement V_j) lorsque p_i (respectivement p_j) a choisit sa valeur de retour comme étant x_i (respectivement x_j).

Supposons par exemple que $V < V'$ (le raisonnement est symétrique pour $V' < V$). Comme p_i a choisi la valeur x_i, cela signifie que $V[id_i] = x_i$, et donc d'après le lemme précédent, $V'[id_i] = x_i$ aussi. Pour que p_j retourne x_i, il doit vérifier $V'[id_j] = x_i$ et $\forall id \neq id_j, V'[id] \neq x_i$ (ligne 21). Mais cela n'est pas vrai, car $V'[id_i] = x_i$, d'où une contradiction.

Il reste la possibilité que $V = V'$. Dans ce cas, un raisonnement similaire s'applique, car $V[id_i]$ et $V[id_j]$ doivent tous deux valoir x_i, et doivent tous deux vérifier que $\forall id \neq id_i$, $V[id] \neq x_i$, ce qui est donc faux.

Enfin, il n'est pas possible que $V \not\leq V'$ (ou $V' \not\leq V$), d'après les lemmes 5.1.1 et 5.1.2. □

Pour prouver que cet algorithme est bien un algorithme de renommage, il reste à démontrer que tout processus correct finit par retourner une valeur. Dans ce but, la première étape est de montrer que la fonction *Valeur_libre* se comporte correctement, dans le lemme 5.1.6.

Pour un tableau donné V, on note $Libre(V)$ l'ensemble des valeurs $x \in [0, n+f-1]$ telles que, $\forall id, V[id].x \neq x$. C'est-à-dire l'ensemble des valeurs qui ne sont proposées par aucun processus.

Lemme 5.1.6. *Quand un processus correct p_i appelle la fonction Valeur_libre ligne 29, il obtient une valeur libre $x \in [0, n+f-1]$*

Démonstration. La fonction *Valeur_libre* prend comme paramètres r et V, et retourne la r-ième valeur de $Libre(V)$, car le tableau *libre* correspond à l'ensemble $Libre(V)$ (une case contenant *vrai* si et seulement si la valeur indice appartient à $Libre(V)$), et la boucle ligne 55 permet d'obtenir la r-ième valeur de ce tableau. Le r est le *rang* de p_i, obtenu par la fonction *Rang_indecis*, et vérifiant $r \geq 1$. En effet, si p_i appelle *Rang_indecis*, alors il vérifie $V_i[id_i].b = 0$ (car il est le seul processus à pouvoir modifier cette valeur, et le fait uniquement avant de retourner ligne 22), et donc $id_i \in indecis$, d'où $rang \geq 1$ car $id_i \leq id_i$ est vrai (ligne 46). De plus, si p_i appelle *Valeur_libre*, alors $r \leq f+1$. Il reste donc à montrer que $|Libre(V)| \geq f+1$. Lorsque p_i appelle *Valeur_Libre*, son tableau V_i vérifie que $V_i[id_i].x = -1$ ou $\exists id_j \neq id_i : V_i[id_i].x = V_i[id_j].x$ (ligne 21). De plus, il existe au plus n cases id telles que $V_i[id] \neq \perp$, correspondant aux n processus. Donc, au plus $n-1$ différentes valeurs $x \in [0, n+f-1]$ sont stockées dans V_i, car soit $V_i[id_i].x = -1 \notin [0, n+f-1]$, soit deux cases différentes ont la même valeur pour le champ x. Donc, $Libre(V_i)$ contient au moins $n+f-(n-1) = f+1$ valeurs distinctes, ce qui suffit à démontrer ce lemme. □

Il reste à démontrer que tout processus correct finit par proposer une valeur qu'il va pouvoir retourner, ce que l'on fait par l'intermédiaire des lemmes suivants, jusqu'au résultat du lemme 5.1.9.

Supposons que, pour une exécution donnée, au moins un processus correct ne retourne pas de valeur x. On note $Indecis$ l'ensemble des processus (corrects ou non) qui ne retournent pas (contenant donc au moins un processus correct par hypothèse). On note ensuite $Continue \subseteq Indecis$ l'ensemble des processus p_i (corrects) qui continuent indéfiniment à proposer de nouvelles valeurs en modifiant à l'infini la valeur de $V[id_i]$ ligne 29.

Lemme 5.1.7. *$Continue \neq \emptyset$.*

Démonstration. Supposons que $Continue = \emptyset$. Cela signifie qu'au bout d'un certain temps, tous les processus de $Indecis$ ne modifient plus les valeurs de leur V_i (et les autres processus aussi une fois qu'ils ont retourné). Dans ces conditions, le système se stabilise avec un unique tableau V contenu dans tous les V_i des processus corrects. Soit p_i le processus correct de $Indecis$ (qui existe) ayant l'identifiant id_i le plus petit parmi les processus corrects de $Indecis$. Le *rang* r de p_i, obtenu via la fonction *Rang_indecis*, est au plus $f+1$. En effet, dans cette fonction la variable *indecis* est construite pour contenir uniquement des processus p_j vérifiant $V[id_j].b = 0$, c'est-à-dire des processus de $Indecis$ (les valeurs

$V[id].b = 1$ des processus ayant terminé finissent par arriver aux processus corrects). Ensuite, *rang* est un entier qui est égal au nombre de processus $p_j \in indecis \subseteq Indecis$ tels que $id_j \leq id_i$, qui est donc au plus $f+1$ par définition de p_i. Comme $r \leq f+1$, le processus p_i atteint alors la ligne 29, et propose alors une nouvelle valeur. Cela signifie que le système n'est pas stabilisé comme supposé, et cette contradiction démontre donc ce lemme. □

Comme $Continue \neq \emptyset$, soit p_0 (d'identifiant id_0) le processus de $Continue$ ayant le plus petit identifiant. L'ensemble des tableaux stables V durant cette exécution est infini, car p_0 continue de modifier sa valeur $V[id_0]$ une infinité de fois. On peut remarquer que, au bout d'un certain temps t, chaque tableau stable V contient exactement k cases $\neq\perp$, pour un certain $k \geq n-f$ fixé, et les valeurs de $V[id_j]$ pour chaque $p_j \notin Continue$ ne changent plus (par définition de $Continue$). En particulier, à partir de t, tous les processus $p_j \notin Indecis$ ont retourné une valeur, et les tableaux stables vérifient $V[id_j].b = 1$. De plus, à partir de t, la variable *indecis* dans la fonction *Rang_indecis* contiendra exactement les identifiants des processus de $Indecis$. Donc, chaque processus p_j qui appelle *Rang_indecis* à partir de t obtient toujours le même rang r_j.

Par définition de $Continue$, p_0 doit vérifier $r_0 \leq f+1$ infiniment souvent pour pouvoir modifier son $V[id_0]$ infiniment souvent. Donc, à partir de t, on a r_0 fixe qui vérifie $r_0 \leq f+1$.

On considère maintenant l'ensemble X_{rest} comme l'ensemble des $x \in [0, n+f-1]$ tels que, à partir de t, $\forall p_j \notin Continue$, $V[id_j].x \neq x$. X_{rest} est un ensemble fixe de $l = n+f-|\mathcal{P}\backslash Continue| \geq n+f-(n-1) = f+1$ valeurs, que l'on peut noter $\{x_1, \ldots, x_l\}$ de sorte que $\forall i \in [1, l-1], x_i < x_{i+1}$. Pour un tableau stable V à partir de t, on a $Libre(V) \subseteq X_{rest}$. De plus, les valeurs que peuvent prendre les $V[id_j]$ des processus $p_j \in Continue$ lors d'une modification postérieure à t sont nécessairement dans l'ensemble X_{rest}.

À partir de t, chaque processus $p_j \in Continue$ va encore modifier son $V[id_j]$. Soit $t' > t$ un instant tel que chaque processus de $Continue$ a effectivement modifié sa valeur depuis t.

Soit un tableau stable V après t'. Pour chaque processus $p_j \in Continue$, $V[id_j] \in X_{rest}$. Considérons la valeur $x_{r_0} \in X_{rest}$ (car $r_0 \leq f+1 \leq l$), on a alors :

Lemme 5.1.8. *Pour tout tableau stable V après t', alors soit $x_{r_0} \in Libre(V)$, soit la seule case vérifiant $V[id].x = x_{r_0}$ est $V[id_0]$.*

Démonstration. Supposons par contradiction que ce lemme est faux. Donc, il existe une case $V[id_j].x = x_{r_0}$ avec $id_j \neq id_0$. Par définition de X_{rest}, cela signifie que $p_j \in Continue$. Soit r le rang de p_j obtenu via *Rang_indecis*, qui vérifie $r > r_0$ par définition de r_0. Lorsque p_j a modifié $V[id_j]$ pour y stocker la valeur x_{r_0}, cela a eu lieu après t (par définition de t'), et cette valeur a été obtenue via la fonction $Valeur\_libre$. Or, cette fonction retourne la r-ième valeur de $Libre(V') \subseteq X_{rest}$. Mais x_{r_0} (si elle était libre) était la r'-ième valeur libre, avec $r' \leq r_0 < r$, ce qui est impossible. □

Pour des raisons similaires à celles évoquées dans la preuve du lemme précédent, pour tout $r < r_0$, et tout tableau stable V postérieur à t', $x_r \in Libre(V)$. Donc, si $V[id_0].x \neq x_{r_0}$, alors $\{x_1, \ldots, x_{r_0}\} \subseteq Libre(V)$, et la valeur retournée par $Valeur\_libre(V, r_0)$ est x_{r_0}. D'où, lorsque p_0 choisit une nouvelle valeur pour $V[id_0]$ (ce qui arrive nécessairement par définition de $Continue$), il choisit x_{r_0}. Donc, il existe un tableau stable vérifiant $V[id_0].x = x_{r_0}$ et $\forall id \neq id_0, V[id].x \neq x_{r_0}$, et tout tableau stable postérieur vérifie aussi cette propriété. Cela signifie que p_0 doit nécessairement retourner la valeur x_{r_0}, et ne fait donc pas partie de $Continue$.

Cette contradiction démontre que tout processus correct retourne nécessairement une valeur.

Lemme 5.1.9. *Tout processus correct retourne une valeur en temps fini.*

Enfin, toute valeur retournée par la fonction Renomme a préalablement été retournée par la fonction $Valeur\_libre$, qui ne produit que des valeurs dans $[0, n+f-1]$, d'où

Théorème 5.1.10. *L'algorithme 5.1 résout le problème de M-renommage, avec $M = n+f$.*

L'article [17] démontre aussi qu'il est impossible, dans ce modèle asynchrone à pannes et par envoi de messages, de résoudre le problème du M-renommage avec $M < n+f$. Ainsi, l'algorithme 5.1 est *optimal* pour ce qui est de la taille de l'espace de noms. Cette démonstration n'est pas retranscrite ici, car ne relevant pas du problème qui nous intéresse.

En effet, comme cela sera démontré plus tard dans le lemme 5.2.1, le problème du renommage ne peut pas (non-trivialement) être résolu en présence d'une majorité de pannes. Avant d'aborder ce problème, on souhaite néanmoins étudier d'autres algorithmes de renommage, ainsi que la version dite *réutilisable* du renommage.

5.1.2 Diviseurs à-un-coup

L'article [78] introduit un nouvel algorithme de renommage (présenté en partie 5.1.3) basé sur une nouvelle forme d'objets partagés appelés *diviseurs*. Ces objets existent eux aussi sous forme *à-un-coup* et *réutilisable*, et la seconde sera présentée dans la partie 5.1.4. Les diviseurs ont été utilisés dans des travaux divers liés au renommage [19, 4, 7, 13], et il est envisageable qu'il soient utiles à d'autres problèmes de par leurs propriétés simples mais intéressantes. On donne maintenant une définition formelle de ces objets :

Les diviseurs à-un-coup disposent d'une unique opération *Divise* qui modifie l'état du diviseur tout en retournant une valeur au processus qui l'a appelée. Un diviseur à-un-coup doit, pour fonctionner correctement, vérifier la condition d'utilisation suivante :

1. Chaque processus ne peut invoquer l'opération Divise sur cet objet qu'au plus *une* fois.

Il vérifie alors les propriétés suivantes :

1. Tout appel à Divise qui termine retourne une valeur parmi $\{bas, droite, stop\}$.
2. Si un processus correct invoque Divise, l'appel termine.
3. Au cours de toute exécution, au plus *un* appel à Divise retourne *stop* au processus qui l'a invoqué.
4. Si au cours d'une exécution, p processus invoquent Divise sur cet objet, au plus $p-1$ de ces appels retournent *bas*, et au plus $p-1$ de ces appels retournent *droite*.

Un corollaire de la dernière propriété est que, si au cours d'une exécution, un unique processus p_i invoque Divise, alors l'appel ne peut pas retourner *bas* ni *droite*, donc si p_i est correct, l'appel retourne nécessairement *stop*.

Un tel objet et ses propriétés peuvent être schématisés comme le montre la figure 5.1. On peut ainsi représenter un diviseur par un bloc dans lequel les processus peuvent entrer (via l'opération Divise), et ensuite ressortir à *droite*, en *bas*, ou rester à l'intérieur via *stop*

L'ensemble de ces propriétés n'est pas sans rappeler l'*exclusion mutuelle*, qui a d'ailleurs servi d'inspiration pour ces diviseurs [69]. Cependant, il est à noter qu'il n'est pas nécessaire qu'un processus obtienne *stop*, et il est donc possible de ne pas avoir de réelle *exclusion mutuelle*, mais dans ce cas les réponses obtenues par les processus sont divisées entre *bas* et *droite*, et tous ne reçoivent pas la même valeur de réponse.

En supposant que chaque processus n'appelle l'opération Divise qu'au plus une fois, il est possible (comme fait dans [78]) d'implémenter simplement un diviseur à l'aide de deux

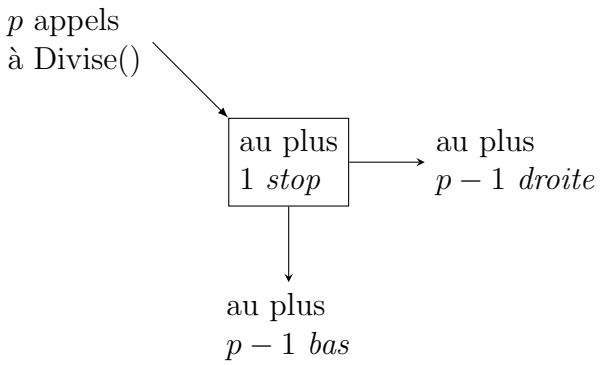

FIGURE 5.1 – Propriétés d'un diviseur

registres atomiques *Porte* et Nom qui sont multi-écrivains et multi-lecteurs. Le registre *Porte* est booléen et ne peut donc contenir comme valeur que $vrai$ ou $faux$.

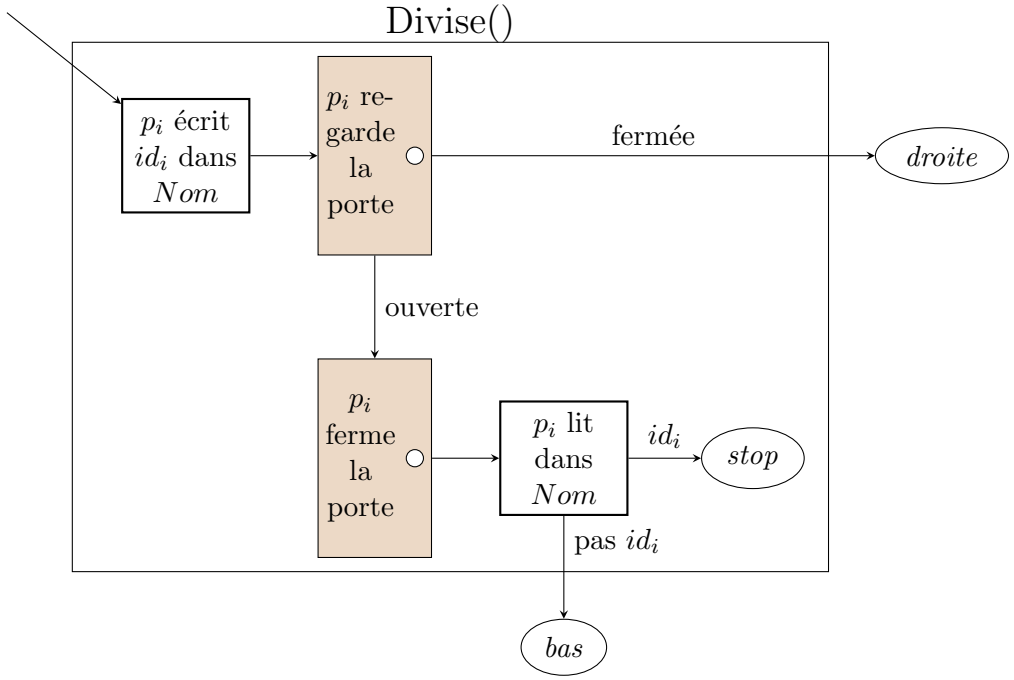

FIGURE 5.2 – Implémentation d'un diviseur à-un-coup

L'algorithme est alors très simple, et peut être illustré par la figure 5.2. Un processus appelant Divise commence par écrire son nom (id_i) dans Nom, puis regarde si la *Porte* est ouverte ou fermée (elle est initialement ouverte). Si la porte est fermée, il retourne *droite*, sinon il ferme la porte. Ensuite, il vérifie si son nom est toujours dans Nom, si c'est le cas il retourne *stop* sinon *bas*.

Algorithme 5.2 Implémentation d'un diviseur à-un-coup (code exécuté par p_i)

1: Initialement, le registre *Porte* contient la valeur $faux$, et la valeur dans Nom n'importe pas.
2: **Fonction** DIVISE(id_i)
3: ECRITURE(id_i) sur le registre Nom ;
4: $b \leftarrow$ LECTURE( ) sur le registre *Porte* ;
5: **si** $b = vrai$ **alors**
6: **retourner** *droite* ;
7: **sinon**

```
 8:      ECRITURE(vrai) sur le registre Porte ;
 9:      n ← LECTURE( ) sur le registre Nom ;
10:      si n = id_i alors
11:          retourner stop ;
12:      sinon
13:          retourner bas ;
```

L'algorithme 5.2 vérifie les propriétés énoncées plus tôt, dès lors que chaque processus n'appelle cette fonction qu'au plus une fois.

Cet algorithme ainsi que les démonstrations qui suivent ne sont pas nouveaux, et restent largement inspirés de [78, 8], mais permettent de mieux comprendre les extensions réalisées dans la partie 5.2.3.

Observation 5.1.11. *Tout appel à Divise qui termine retourne une valeur parmi* $\{bas, droite, stop\}$.

Lemme 5.1.12. *Si un processus correct invoque Divise, l'appel termine.*

Démonstration. Si un processus correct les invoque, les opérations de Lecture et d'Ecriture terminent, donc la fonction Divise termine. □

Lemme 5.1.13. *Si au cours d'une exécution, p processus invoquent Divise sur cet objet, au plus* $p-1$ *de ces appels retournent* droite.

Démonstration. Pour qu'un processus retourne *droite*, il lui faut lire *vrai* dans le registre *Porte* (ligne 4), mais celui-ci ne prend pour valeur *vrai* qu'après qu'un processus ait atteint la ligne 8. Or, un tel processus qui écrit *vrai* ne peut plus retourner *droite*, ce qui prouve bien la propriété. □

Lemme 5.1.14. *Si au cours d'une exécution, p processus invoquent Divise sur cet objet, au plus* $p-1$ *de ces appels retournent* bas.

Démonstration. Pour qu'un processus retourne *bas*, il lui faut lire une valeur dans *Nom* qui n'est pas son propre identifiant id_i. On supposera que tous les processus qui invoquent Divise ne tombent pas en panne durant cet appel, car dans le cas contraire le lemme est vérifié. On considère le dernier processus à retourner de l'appel à Ecriture ligne 3, ainsi que tous les processus dont l'appel correspondant y est concurrent (de tels processus existent grâce au nombre fini de processus du système et à la limite d'au plus un appel par processus). Du point de vue de la linéarisabilité, un de ces appels est logiquement considéré comme le *dernier*, c'est-à-dire que toute Lecture postérieure à cet appel retournera nécessairement la valeur écrite lors de cet appel. Notons p_i le processus qui a exécuté ce dernier appel, en écrivant id_i. Comme cet appel est le *dernier* d'un point de vue logique du registre, il lira id_i à la ligne 9, et ne pourra donc pas retourner *bas*, ce qui démontre ce lemme. □

Lemme 5.1.15. *Au cours de toute exécution, au plus* un *appel à Divise retourne* stop *au processus qui l'a invoqué.*

Démonstration. Supposons qu'au moins 2 processus, notés p_1 et p_2, retournent *stop*, et considérons que p_1 appelle l'opération Lecture ligne 9 avant p_2. Comme p_1 retourne *stop*, cette Lecture, notée L_1, retourne id_1. De même, la Lecture correspondante (L_2) exécutée par p_2 retourne id_2. Ces valeurs id_1 et id_2 ont été écrites dans le registre lors des opérations d'Ecriture E_1 et E_2 exécutées par p_1 et p_2 ligne 3. On sait que E_1 *précède* L_1 et L_2, et E_2 précède L_2. Si E_2 précède L_1, alors les 2 lectures commencent après que les 2 écritures soient terminées, et ne peuvent donc pas retourner les 2 valeurs différentes écrites lors de ces écritures id_1 et id_2, ce qui aboutit à une contradiction. Si E_2 ne précède pas L_1, alors

E_2 termine après que L_1 commence, donc en particulier après que le processus p_1 ait écrit *vrai* dans le registre *Porte* (ligne 8). Donc lorsque le processus p_2 effectuera sa Lecture sur ce registre *Porte* (ligne 4), cette opération retournera *vrai* (car aucun processus n'écrit $faux$ dans ce registre), et donc p_2 retournera *droite*, ce qui est une contradiction. Ainsi, p_1 et p_2 ne peuvent pas tous deux retourner *stop*. □

Ces lemmes permettent donc de démontrer que, par définition :

Théorème 5.1.16. *L'algorithme 5.2 implémente un diviseur à-un-coup.*

5.1.3 Renommage à base de diviseurs

Il est possible d'utiliser des diviseurs afin de résoudre le problème du renommage, raison pour laquelle les diviseurs ont été introduits dans [78].

L'algorithme de renommage présenté dans ce même article utilise une grille de diviseurs sous la forme d'un tableau à 2 dimensions, dont au plus $n(n+1)/2$ sont susceptibles d'être utilisés.

Cette grille prend une forme triangulaire, et on associe à chaque diviseur de la grille un entier unique grâce à la formule $x = (a+b)*(a+b+1)/2+b$. La figure 5.3 représente une telle grille pour des diviseurs dont les coordonnées $\langle a, b\rangle$ vérifient $a + b < 5$. Les processus entrent dans la grille en appelant Divise sur un diviseur, en commençant par le diviseur 0 (en vert sur la figure). De manière assez intuitive, les processus obtenant *droite* (respectivement *bas*) d'un diviseur d, appelent ensuite Divise sur le diviseur situé à droite (respectivement en bas) de d dans la grille. Enfin, lorsqu'un processus obtient *stop*, il retourne pour le renommage la valeur x identifiant le diviseur en question.

Algorithme 5.3 Renommage à-un-coup à base de diviseurs. (code exécuté par p_i)

```
1:  Initialisation
2:      GD est un tableau à 2 dimensions de diviseurs à-un-coup, dont on peut accéder à
        un diviseur via GD[a, b], avec a, b ∈ ℕ et a + b < n.
3:      a_i ← 0 ; b_i ← 0 ;
4:  Fonction RENOMME(id_i)
5:      a_i ← 0 ; b_i ← 0 ;
6:      répéter
7:          d_i ← DIVISE(id_i) exécuté sur le diviseur GD[a_i, b_i] ;
8:          si d_i = bas alors
9:              b_i ← b_i + 1 ;
10:         sinon si d_i = droite alors
11:             a_i ← a_i + 1 ;
12:     jusqu'à ce que d_i = stop
13:     x_i ← (a_i + b_i) * (a_i + b_i + 1)/2 + b_i ;
14:     retourner x_i ;
```

La valeur retournée, $x_i = (a_i + b_i) * (a_i + b_i + 1)/2 + b_i$, est un nombre unique dans le sens où deux paires $\langle a, b\rangle \neq \langle a', b'\rangle$ auront un x différent (si $a, a', b, b' \in \mathbb{N}$). En effet, si $a + b = a' + b'$, la différence entre les x est exactement $b - b'$, et dans le cas contraire, si (sans perdre en généralités) $a + b > a' + b'$, alors $a + b - 1 \geq a' + b'$, et $x(a, b) - x(a', b') \geq (a + b) * (a + b + 1)/2 - (a + b - 1) * (a + b)/2 + b - b' \geq a + b - b' \geq a + b - (a' + b') > 0$. En somme, on a :

Observation 5.1.17. *Chaque valeur x_i est associée à une unique paire $\langle a_i, b_i\rangle$ et donc à un unique diviseur de la grille GD.*

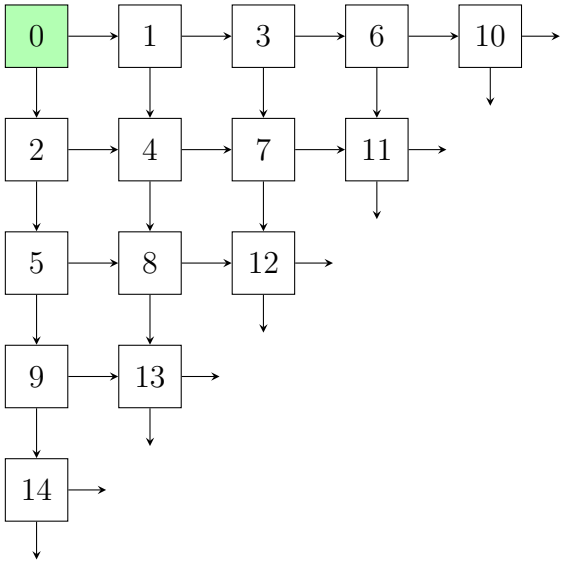

FIGURE 5.3 – Grille de diviseurs pour le renommage, de largeur 5

De plus, on remarquera que, si $a_i + b_i \leq n - 1$, alors $x_i \leq n(n-1)/2 + n - 1 < n(n+1)/2$. Donc, dans ces conditions, les valeurs de retour sont dans l'ensemble d'entiers $[0, n(n+1)/2 - 1]$.

Les lemmes qui suivent démontrent que cet algorithme 5.3 est bien un algorithme de M-renommage, avec $M = n(n+1)/2$. On suppose que la fonction *Renomme* est correctement utilisée, c'est-à-dire qu'un processus p_i ne peut invoquer *Renomme* qu'au plus une fois. L'algorithme et les démonstrations sont largement inspirés des résultats de [78, 8], mais sont présentés ici afin de mieux comprendre leurs versions étendues de la partie 5.2.4.

Lemme 5.1.18. *Si deux processus p_i et p_j ont obtenus deux valeurs x_i et x_j, alors $x_i \neq x_j$.*

Démonstration. Un processus n'obtient une valeur x que lorsqu'il obtient la réponse *stop* à son appel à *Divise* sur le x-ième diviseur de la grille, ligne 7. Donc, si $x_i = x_j$, cela signifie que p_i et p_j ont tous deux obtenu la réponse *stop* lors de leur appel à *Divise* sur le même diviseur. Cela contredit la propriété des diviseurs à-un-coup, et est donc impossible. □

Il reste à montrer que $a_i + b_i \leq n$, ce qui suffit en particulier à montrer que la fonction *Renomme* termine si appelée par un processus correct, car *Divise* termine aussi et $a_i + b_i$ est incrémenté de 1 à chaque tour de boucle (sauf lorsque *stop* est retourné, ce qui termine la fonction).

Pour un certain $\langle a, b \rangle$ on parlera de la sous-grille $G_{a,b}$ comme la grille des diviseurs $GD[c, d]$ avec $c \geq a$ et $d \geq b$. En particulier, un processus exécutant *Renomme* qui accède au diviseur $GD[a, b]$ ne pourra à partir de ce moment appeler que des diviseurs de $G_{a,b}$. Au cours d'une exécution, on dira que la sous-grille $G_{a,b}$ *contient* un processus p_i si ce processus a appelé *Divise* sur un des diviseurs de cette sous-grille. Si p_i retourne alors une valeur, ce sera nécessairement celle associée à un diviseur de cette sous-grille.

Lemme 5.1.19. *Pour toute exécution, $\forall a, b \in \mathbb{N}$, la sous-grille $G_{a,b}$ contient au plus $n - (a + b)$ processus.*

Démonstration. On démontre cette propriété par induction sur $a + b$. Comme le système contient n processus, cette propriété est vraie pour la sous-grille $G_{0,0}$ qui est exactement la grille GD. Supposons maintenant que, pour un certain $d \in \mathbb{N}$ fixé, toute sous-grille

$G_{a',b'}$ avec $a' + b' \leq d$ vérifie cette propriété, et considérons une sous-grille $G_{a,b}$ avec $a + b = d + 1$. Supposons par contradiction que cette sous-grille contienne au moins $n - (a + b) + 1$ processus. Comme tous les processus de cette sous-grille sont contenus dans $G_{a-1,b}$ et $G_{a,b-1}$ (au moins 1 de ces 2 sous-grilles existe, car $a + b \geq 1$), d'où on sait par induction que $G_{a,b}$ contient au plus $n - (a + b) + 1$ processus. Cela signifie en particulier que tous les processus contenus dans $G_{a-1,b}$ et $G_{a,b-1}$ sont contenus dans $G_{a,b}$. Pour arriver dans $G_{a,b}$, ces processus ont nécessairement évolué dans la grille, diviseur par diviseur, et en incrémentant à chaque fois une seule des coordonnées. Donc, un processus p_i est arrivé dans un premier diviseur $GD[a_i, b_i]$ de cette sous-grille vérifiant $a_i = a$ et $b_i \geq b$, ou $b_i = b$ et $a_i \geq a$. En particulier, il a atteint un tel diviseur en obtenant *droite* du diviseur $GD[a-1, b_i]$ avec $b_i \geq b$ ou *bas* du diviseur $GD[a_i, b-1]$ avec $a_i \geq a$. Supposons qu'au moins un processus p_i vérifie le premier cas, et considérons alors P_m les processus entrant dans la sous-grille $G_{a,b}$ via le diviseur $GD[a-1, b_i]$ avec le b_i le plus grand, noté b_m. Donc, en particulier, tous les processus de P_m obtiennent *droite* du diviseur $GD[a-1, b_m]$. Comme tous les processus contenus dans $G_{a-1,b}$ sont aussi contenus dans $G_{a,b}$, cela signifie en particulier que tous les processus accédant à $GD[a-1, b_m]$ entrent dans la sous-grille $G_{a,b}$. Or, comme les processus entrant dans $G_{a,b}$ via un $GD[a-1, b_i]$ le font avec $b_i \leq b_m$ (par définition de b_m), cela signifie que tous les processus accédant à $GD[a-1, b_m]$ obtiennent *droite*, car si ils obtiennent *bas* cela contredit la définition de b_m, et si ils obtiennent *stop* ils n'entreront pas dans $G_{a,b}$. Donc, tous les processus entrant appelant $Divise[a-1, b_m]$ obtiennent *droite*, ce qui n'est pas possible. Donc, aucun processus n'entre dans $G_{a,b}$ via un diviseur $GD[a-1, b_i]$. De même, en considérant les processus entrant dans $G_{a,b}$ via un diviseur $GD[a_i, b-1]$, on montre que pour un certain diviseur $GD[a_m, b-1]$, tous les processus (dont au moins 1) retournent *bas*, ce qui est impossible. Donc, les processus contenus dans $G_{a-1,b}$ et $G_{a,b-1}$ ne sont pas tous contenus dans $G_{a,b}$. D'où on a au plus $n - (a + b)$ processus contenus dans $G_{a,b}$. □

Cela permet donc de conclure qu'aucun processus ne peut accéder à une sous-grille $G_{a,b}$ avec $a + b \geq n$. Ce qui permet d'affirmer :

Théorème 5.1.20. *L'algorithme 5.3 est un algorithme de M-renommage à-un-coup, avec $M = n(n+1)/2 = O(n^2)$.*

5.1.4 Diviseurs réutilisables

L'article [78] présente aussi une autre version du renommage appelée renommage *réutilisable*, ainsi que des objets de diviseurs *réutilisables* associés. Avant d'entrer dans les détails de cette version étendue du problème, il semble plus simple de se pencher sur la version étendue de cet objet partagé qu'est le diviseur. Cette simple extension peut une fois de plus être utile pour le renommage [78, 8], mais pourraient aussi être utilisés pour d'autres problèmes grâce à leurs propriétés simples mais élégantes. La principale différence avec leur version *à-un-coup* est que ces objets peuvent être utilisés à plusieurs reprises au cours d'un algorithme, comme cela est visible au travers de la définition suivante :

Les diviseurs réutilisables disposent d'une opération Divise similaire à celle des diviseurs à-un-coup, et ont aussi une opération Relache, qui modifie l'état de cet objet sans retourner de valeur au processus qui l'a appelée. Un diviseur réutilisable doit être utilisé en respectant certaines contraintes :

1. Un processus ne peut appeler Relache que si son dernier appel à Divise a retourné *stop*, et qu'il n'a pas invoqué Relache depuis le retour de cette opération Divise.
2. Un processus ne peut pas appeler Divise si son dernier appel à Divise a retourné *stop* et qu'il n'a pas encore invoqué Relache depuis cette opération Divise.

Et il vérifie alors les propriétés :

1. Tout appel à Divise qui termine retourne une valeur dans $\{bas, droite, stop\}$.
2. Si un processus correct invoque Divise ou Relache, cette opération termine.
3. À tout instant, au plus un processus p_i a obtenu *stop* lors d'un appel à Divise et n'a pas appelé l'opération Relache depuis.
4. Durant tout intervalle de temps qui est une *période occupée* (définie ci-dessous), les appels à Divise ne retournent pas tous *droite*.
5. Durant tout intervalle de temps qui est une *période occupée* comprenant un nombre *fini* d'appels à Divise, tous ne retournent pas tous *bas*.

Un diviseur est *occupé* à un instant t si une opération Divise est en cours (a été invoqué et n'a pas terminé) sur le diviseur, ou un processus a obtenu *stop* et l'appel correspondant à Relache n'a pas encore été invoqué ou n'a pas terminé. Une *période occupée* est alors un intervalle de temps durant lequel le diviseur est occupé à tout instant, et qui est *maximal* (c'est-à-dire que tout autre intervalle de temps contenant cette période contient aussi un instant durant lequel le diviseur n'est pas occupé). On peut noter qu'une telle période peut être infinie, notamment si un processus tombe en panne lors d'un appel à une opération, et que cette opération ne termine jamais. On dira qu'un processus a *capturé* le diviseur s'il a obtenu *stop* et n'a pas encore appelé la fonction Relache depuis.

La figure 5.4 présente un exemple d'exécution sur un diviseur réutilisable, et permet de mieux visualiser la notion de période occupée.

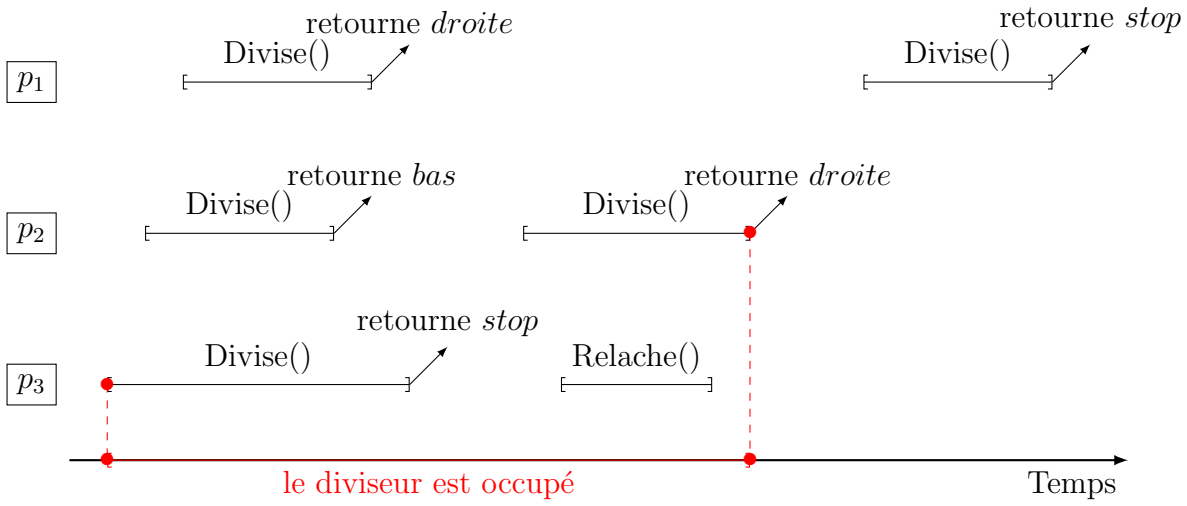

FIGURE 5.4 – Exécution possible sur un diviseur réutilisable

En supposant que l'objet est correctement utilisé, et que les propriétés correspondantes sont donc vérifiées, on peut implémenter un diviseur réutilisable de manière similaire à la simulation précédente (algorithme 5.2). Une différence notable est qu'on utilise n registres $Porte_i$ (booléens) qui sont chacun mono-écrivain multi-lecteurs, associés aux processus correspondants p_i, au lieu d'un unique registre *Porte* multi-écrivains.

Le fonctionnement global est encore identique, représenté par le schéma 5.5, à la différence suivante : À chaque fois qu'un processus passe la porte et la *referme*, il ne le fait plus anonymement comme dans l'algorithme 5.2, mais il pose plutôt un *cadenas* personnel sur cette porte. Ensuite, lorsqu'un processus devrait obtenir *bas*, il retire son *cadenas*, ce qui a pour effet d'ouvrir à nouveau la porte, à condition que tous les cadenas soient ainsi retirés. Enfin, un processus obtenant *stop* ne retire son cadenas de la porte qu'en appelant la fonction Relache, ce qui permet de garantir que la porte reste fermée durant ce temps.

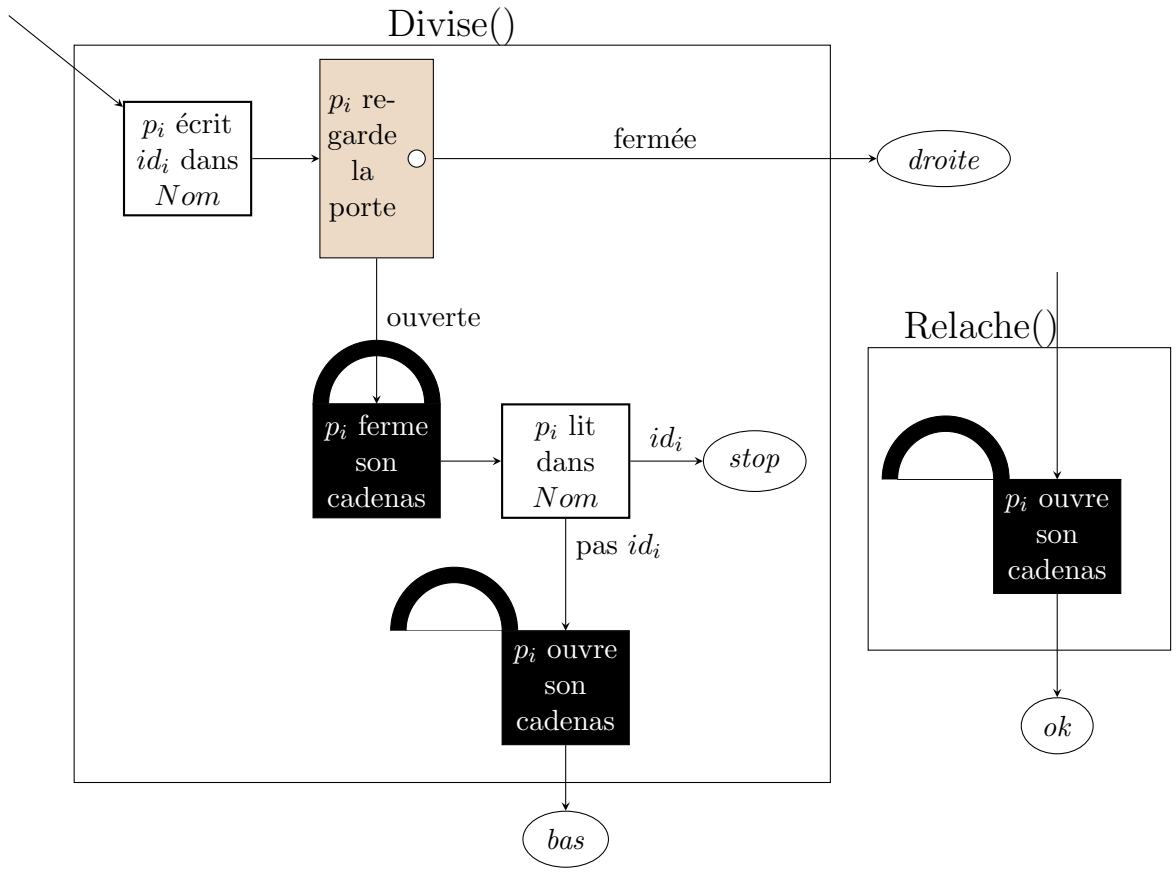

FIGURE 5.5 – Implémentation d'un diviseur réutilisable

Algorithme 5.4 Implémentation d'un diviseur réutilisable (code exécuté par p_i)

1: Initialement, les registres $Porte_j$ contiennent la valeur $faux$, et la valeur dans Nom n'importe pas.
2: **Fonction** DIVISE(id_i)
3: ECRITURE(id_i) sur le registre Nom ;
4: $b \leftarrow faux$;
5: **pour chaque** processus p_j **faire**
6: $b \leftarrow b \vee$(LECTURE( ) sur le registre $Porte_j$) ; ▷ $\vee$ est le *ou* logique
7: **si** $b = vrai$ **alors**
8: **retourner** $droite$;
9: **sinon**
10: ECRITURE($vrai$) sur le registre $Porte_i$;
11: $n \leftarrow$ LECTURE( ) sur le registre Nom ;
12: **si** $n = id_i$ **alors**
13: **retourner** $stop$;
14: **sinon**
15: ECRITURE($faux$) sur le registre $Porte_i$;
16: **retourner** bas ;
17: **Fonction** RELACHE(id_i)
18: ECRITURE($faux$) sur le registre $Porte_i$;

On peut alors montrer que cet algorithme 5.4 vérifie bien les propriétés définies plus tôt.

Une fois de plus, l'algorithme comme les démonstrations suivantes sont fortement inspirés des [78, 8], mais permettent d'aborder plus aisément ceux de la partie 5.3.2.

Observation 5.1.21. *Tout appel à Divise qui termine retourne une valeur dans* $\{bas, droite, stop\}$.

Lemme 5.1.22. *Si un processus correct invoque Divise ou Relache, cette opération termine.*

Démonstration. Chaque appel à Ecriture ou Lecture par un processus correct termine, et la seule boucle dans l'opération Divise (ligne 5) n'effectue que n itérations. Donc chaque appel à Divise ou à Relache par un processus correct termine. □

On montre alors le lemme suivant, utile dans la démonstration du lemme 5.1.24

Lemme 5.1.23. *À tout instant où le diviseur n'est pas occupé (en particulier au début de chaque période occupée), les registres* $Porte_j$ *contiennent tous la valeur* $faux$.

Démonstration. Cette propriété est initialement vraie. De plus, à chaque fois qu'un appel à Divise par un processus p_i termine, il y a trois possibilités. Si cet appel retourne *droite*, alors comme p_i n'a pas atteint la ligne 10, le registre $Porte_i$ contient toujours sa valeur précédente (initialement $faux$). Si cet appel retourne *bas*, alors le registre $Porte_i$ contient $faux$ (ligne 15). Si cet appel retourne *stop*, alors $Porte_i$ contient la valeur $vrai$ (ligne 10). Mais si p_i obtient *stop*, le diviseur reste *occupé* jusqu'à ce que p_i appelle Relache et que cet appel termine. Or, après un appel à Relache, on a $Porte_i$ qui contient $faux$ (ligne 18). Donc, lorsque le diviseur n'est pas occupé, la valeur de $Porte_i$ est nécessairement $faux$. Comme cela est vrai pour tout processus p_i, ce lemme est ainsi démontré. □

Lemme 5.1.24. *Durant tout intervalle de temps qui est une* période occupée *, les appels à Divise ne retournent pas tous* droite.

Démonstration. Au début d'une période occupée, tous les registres $Porte_j$ contiennent $faux$. De plus, pour retourner *droite*, un processus p_i doit lire au moins un $vrai$ parmi les registres $Porte_j$ (ligne 6). Or, pour qu'un des registres $Porte_j$ prenne pour valeur $vrai$, il faut que le processus p_j correspondant atteigne la ligne 10. Lors de cet appel, p_j ne peut alors plus retourner *droite*, ce qui démontre bien ce lemme. □

Lemme 5.1.25. *Durant tout intervalle de temps qui est une* période occupée *et qui comprend un nombre* fini *d'appels à Divise, tous ne retournent pas tous* bas.

Démonstration. Pour qu'un processus retourne *bas*, il lui faut lire une valeur dans Nom qui n'est pas son propre identifiant id_i. On supposera que tous les processus qui invoquent Divise ne tombent pas en panne durant cet appel, car dans le cas contraire le lemme est vérifié. On considère l'exécution de Divise qui est la dernière a retourner de l'appel à Ecriture ligne 3, ainsi que toutes les invocations de Divise dont l'appel correspondant y est concurrent (qui sont en nombre fini). Du point de vue de la linéarisabilité, un de ces appels est logiquement considéré comme le *dernier*, c'est-à-dire que toute Lecture postérieure à cet appel retournera nécessairement la valeur écrite lors de cet appel. Notons p_i le processus qui a exécuté ce dernier appel, en écrivant id_i. Comme cet appel est le *dernier* d'un point de vue logique du registre, il lira id_i à la ligne 11, et ne pourra donc pas retourner *bas*, ce qui démontre ce lemme. □

Lemme 5.1.26. *À tout instant, au plus un processus* p_i *a obtenu* stop *lors d'un appel à Divise et n'a pas appelé l'opération Relache depuis.*

Démonstration. Supposons qu'à un instant t, au moins 2 processus, notés p_1 et p_2, aient obtenu *stop* et n'aient pas encore appelé Relache depuis, et considérons que p_1 appelle l'opération Lecture ligne 11 avant p_2. Comme p_1 obtient *stop*, cette Lecture, notée L_1, retourne id_1. De même, la Lecture correspondante (L_2) exécutée par p_2 retourne id_2. Ces

valeurs id_1 et id_2 ont été écrites dans le registre lors des opérations d'Ecriture E_1 et E_2 exécutées par p_1 et p_2 ligne 3. On sait que E_1 *précède* L_1 et L_2, et E_2 précède L_2. Si E_2 précède L_1, alors les 2 lectures commencent après que les 2 écritures soient terminées, et ne peuvent donc pas retourner les 2 valeurs différentes écrites lors de ces écritures id_1 et id_2, ce qui aboutit à une contradiction. Si E_2 ne précède pas L_1, alors E_2 termine après que L_1 commence, donc en particulier après que le processus p_1 ait écrit *vrai* dans le registre $Porte_i$ (ligne 10). Donc lorsque le processus p_2 effectuera sa Lecture sur ce registre $Porte_i$ (ligne 6), cette opération retournera *vrai* (car p_1 n'a pas encore appelé Relache depuis cette écriture, et comme il est le seul processus a pouvoir écrire dans ce registre, $Porte_i$ contient toujours *vrai*), et donc p_2 retournera *droite*, ce qui est une contradiction. Ainsi, p_1 et p_2 ne peuvent pas tous deux avoir obtenu *stop* et pas encore appelé Relache. □

Les lemmes précédents permettent donc de déduire, par définition :

Théorème 5.1.27. *L'algorithme 5.4 implémente un diviseur réutilisable.*

5.1.5 Renommage réutilisable

Le renommage *réutilisable* a été introduit dans [78] pour apporter une nouvelle version, plus générale, du renommage *à-un-coup*. La différence entre les deux versions est qu'ici, les processus peuvent exécuter plusieurs fois le renommage, et obtiennent à chaque fois un nouveau nom (ou valeur) qui peut être différent. Cependant, entre deux participations au renommage, un processus doit *libérer* la valeur précédemment acquise, afin de permettre à d'autres processus (ou lui-même à nouveau) d'obtenir cette valeur. Ainsi, une telle version s'approche plus de problèmes réels comme l'allocation de ressources, car ces ressources ne sont en général utilisées que pendant un temps limité, et d'autres processus devraient ensuite pouvoir y accéder.

Plus formellement, le renommage *réutilisable* doit être correctement utilisé via les critères suivants :

1. Initialement, chaque processus p_i a un identifiant unique id_i.
2. Les processus peuvent chercher à obtenir une valeur et libérer une valeur précédemment obtenue, et ce autant de fois que souhaité.
3. Un processus ne peut chercher à obtenir une valeur que s'il a libéré la dernière valeur qu'il a obtenue.
4. Un processus ne peut libérer une valeur que s'il l'a précédemment obtenue et pas encore libérée depuis.

Et il vérifie alors les propriétés suivantes :

1. Chaque processus correct p_i qui désire obtenir une valeur x_i doit l'obtenir en temps fini.
2. Chaque processus correct p_i qui désire libérer une valeur doit le faire en temps fini.
3. Si à un instant t deux processus distincts p_i et p_j ont obtenu deux valeurs x_i et x_j et ne les ont pas encore libérées, alors $x_i \neq x_j$.

On dira qu'un processus p_i qui a obtenu une valeur x_i mais ne l'a pas encore libérée a *capturé* cette valeur. La dernière propriété peut donc se formuler comme suit : à tout instant, chaque valeur x est capturée par au plus un processus.

Un algorithme de renommage réutilisable dispose donc de deux opérations Renomme et Libere, qui doivent terminer (si appelées par un processus correct). Une valeur x est alors capturée entre le moment où Renomme retourne x, et celui où le processus appelle Libere. Un exemple d'exécution est présenté dans la figure 5.6.

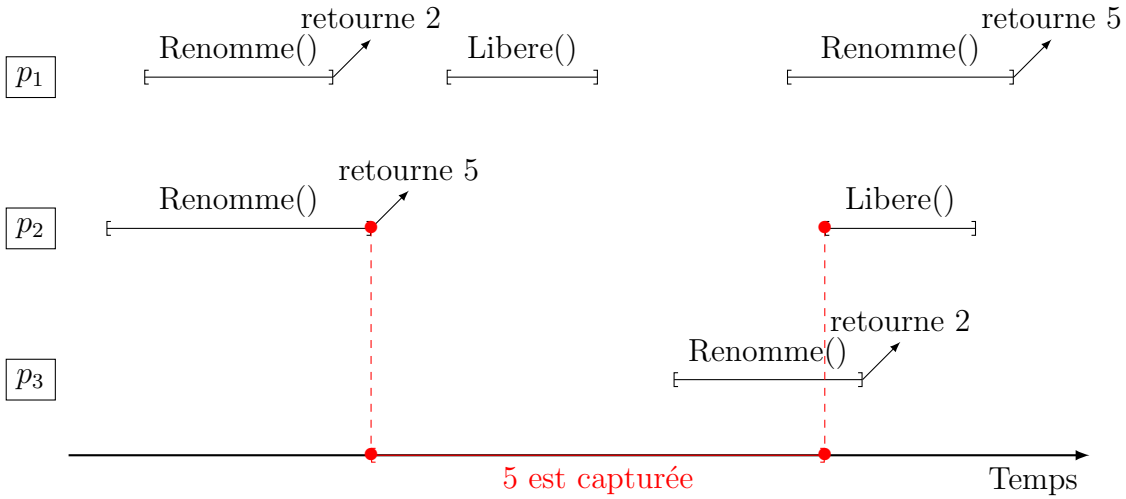

FIGURE 5.6 – Exécution possible d'un renommage réutilisable

Un algorithme permettant de réaliser un renommage *réutilisable* est alors possible, basé sur une grille de diviseurs similaire à celle de l'algorithme de renommage *à-un-coup* précédent, mais utilisant des diviseurs *réutilisables* à la place. C'est d'ailleurs la raison pour laquelle ces objets ont été introduits.

De plus, on utilisera une grille *carrée*, notamment pour ce qui est de l'attribution de noms aux diviseurs, via la formule $x = a*2n+b$. Une telle grille est présentée dans l'article [8]. En supposant que les seuls diviseurs auxquels on accède sont ceux de coordonnées $\langle a, b\rangle$ avec $a < 2n$ et $b < 2n$, cela donne la grille de la figure 5.7.

Une fois de plus, les processus évoluent dans la grille en commençant par le diviseur 0, puis avancent vers la droite ou le bas selon les réponses obtenues aux appels à Divise, jusqu'à obtenir *stop* sur un diviseur. En obtenant *stop*, un processus capture le diviseur et le nom associé x, puis il libère ce nom en appelant Relache sur ce même diviseur. Enfin, si un processus qui a obtenu une valeur et l'a libéré décide de participer à nouveau au renommage, il retourne au début de la grille (diviseur 0).

Algorithme 5.5 Renommage réutilisable à base de diviseurs. (code exécuté par p_i)

1: **Initialisation**
2: GD est un tableau à 2 dimensions de diviseurs *réutilisables*, dont on peut accéder à un diviseur via $GD[a, b]$, avec $a, b \in \mathbb{N}$ et $a < 2n$ et $b < 2n$.
3: $a_i \leftarrow 0$; $b_i \leftarrow 0$;
4: **Fonction** RENOMME(id_i)
5: $a_i \leftarrow 0$; $b_i \leftarrow 0$;
6: **répéter**
7: $d_i \leftarrow$ DIVISE(id_i) exécuté sur le diviseur $GD[a_i, b_i]$;
8: **si** $d_i = bas$ **alors**
9: $b_i \leftarrow b_i + 1$;
10: **sinon si** $d_i = droite$ **alors**
11: $a_i \leftarrow a_i + 1$;
12: **jusqu'à ce que** $d_i = stop$
13: $x_i \leftarrow a_i * (2n) + b_i$;
14: **retourner** x_i ;
15: **Fonction** LIBERE(id_i)
16: exécute RELACHE(id_i) sur le diviseur $GD[a_i, b_i]$;

On démontre maintenant que cet algorithme permet bien de résoudre le M-renommage

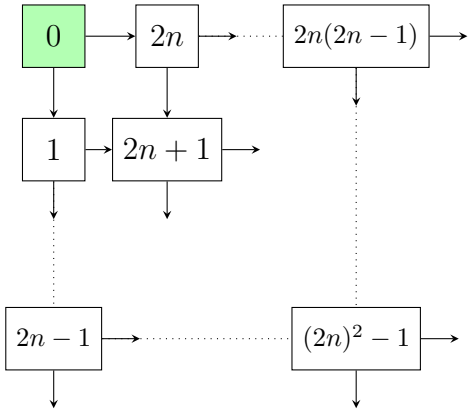

FIGURE 5.7 – Grille de diviseurs pour le renommage réutilisable

réutilisable avec $M = (2n)^2 = O(n^2)$.

Cette fois, chaque diviseur $\langle a, b\rangle$ avec $0 \leq a < 2n$ et $0 \leq b < 2n$ est associé à une unique valeur $x \in [0, (2n)^2 - 1]$, via la formule $x = a * (2n) + b$. L'espace des noms ainsi utilisé pour ce renommage est alors $[0, (2n)^2 - 1]$, pour une taille de $(2n)^2$.

On suppose que les fonctions *Renomme* et *Libere* sont correctement utilisées, c'est-à-dire qu'un processus p_i ne peut invoquer *Renomme* que s'il a libéré toute valeur précédemment obtenue, et qu'il ne peut invoquer *Libere* que s'il a précédemment obtenu une valeur via *Renomme* et ne l'a pas encore libérée.

Une fois de plus, cet algorithme et les démonstrations suivantes sont inspirés de [78, 8], mais sont utiles à la compréhension de la notion de renommage *réutilisable* et aux idées présentées dans la partie 5.3.3.

Le premier lemme démontre l'unicité des valeurs capturées, à condition que les processus restent dans la grille carrée de largeur $2n$, mais cette condition sera justifiée dans le lemme 5.1.31.

Lemme 5.1.28. *En supposant qu'on vérifie toujours $a < 2n$ et $b < 2n$, si à un instant t, deux processus p_i et p_j ont capturé deux valeurs x_i et x_j, alors $x_i \neq x_j$.*

Démonstration. Un processus ne capture une valeur x que lorsqu'il obtient la réponse *stop* à son appel à *Divise* sur le x-ième diviseur de la grille, ligne 7. De plus, un tel processus n'appelle *Relache* sur ce diviseur que lorsqu'il choisit de libérer sa valeur via la fonction *Libere*. Donc, si $x_i = x_j$, cela signifie que p_i et p_j ont tous deux obtenu la réponse *stop* lors de leur appel à *Divise* sur le même diviseur, alors qu'aucun des deux n'a encore appelé *Relache* depuis. Cela contredit la propriété des diviseurs réutilisables, et est donc impossible. □

Lemme 5.1.29. *Tout appel à Libere par un processus correct termine.*

Démonstration. Cela vient directement du fait que tout appel à *Relache* par un processus correct termine. □

Il reste à démontrer que tout appel à *Renomme* par un processus correct termine et que $a_i < 2n$ et $b_i < 2n$. Il suffit pour cela de montrer qu'aucun processus ne peut vérifier $a_i \geq 2n$ ou $b_i \geq 2n$, car chaque appel à *Divise* par un processus correct termine en temps fini.

Au cours d'un appel à *Renomme*, un processus p_i évoluera dans la grille de diviseurs DG en commençant au diviseur $\langle 0, 0\rangle$, et en se déplaçant vers la droite ou vers le bas par l'incrémentation d'une des deux coordonnées.

On notera L_ℓ (respectivement C_ℓ) la ℓ-ième ligne (colonne) de la grille, c'est-à-dire l'ensemble des diviseurs de coordonnées $\langle \ell, b\rangle$ (respectivement $\langle a, \ell\rangle$) avec un $a, b \in \mathbb{N}$ quelconque.

À un instant t d'une exécution, on dira qu'une ligne L_ℓ (respectivement colonne C_ℓ) est *occupée* si au moins un des diviseurs de cette ligne (colonne) est dans une période occupée à cet instant, ou si un processus p_i a obtenu la réponse *droite* (respectivement *bas*) d'un diviseur de cette ligne (colonne) et n'a pas encore appelé *Divise* sur le diviseur suivant de cette même ligne (colonne). À un instant t, une ligne (colonne) occupée et telle qu'au moins un de ses diviseurs a retourné *stop* au cours de cette période occupée, et que l'appel correspondant à *Relache* a terminé, est dite *utilisée*. De plus, on dira qu'une ligne L_ℓ (colonne C_ℓ) *contient* un processus p_i à un instant t si ce processus est *occupé* avec un des diviseurs de L_ℓ (respectivement C_ℓ), ou si p_i a obtenu une réponse *bas* (respectivement *droite*) d'un diviseur de $L_{\ell-1}$ (respectivement $C_{\ell-1}$), et n'a pas encore appelé *Divise* sur le diviseur correspondant de L_ℓ (C_ℓ). Un tel processus est alors *contenu* par la ligne (colonne) en question.

Le lemme suivant est utile pour la démonstration du lemme 5.1.31.

Lemme 5.1.30. *Lors de toute période occupée* finie *d'une ligne L_ℓ (respectivement colonne C_ℓ), au moins un appel à l'un de ses diviseurs retourne stop.*

Démonstration. Supposons par contradiction qu'il existe une telle ligne (colonne) et une période occupée finie de cette dernière telle qu'aucun des appels ne retourne *stop*. On peut remarquer que, par définition d'une période occupée pour une ligne (colonne), aucun processus ne peut tomber en panne lors d'un appel à *Divise* sur un des diviseurs de cette ligne (colonne), ou entre l'obtention de *droite* (respectivement *bas*) et l'appel à *Divise* sur le diviseur suivant de cette ligne (colonne). Comme il y a un nombre fini de tels appels, considérons l'ensemble des appels effectués sur le diviseurs le plus à droite (respectivement en bas) de cette ligne (colonne), noté $\langle \ell, d\rangle$ (respectivement $\langle d, \ell\rangle$). Alors, comme cette période occupée est finie, ces appels ne retournent pas tous *bas* (ni tous *droite*), et comme aucun ne retourne *stop*, cela signifie qu'au moins 1 retourne *droite* (*bas*). Donc, au moins un processus accède au diviseur $\langle \ell, d+1\rangle$ (respectivement $\langle d+1, \ell\rangle$), ce qui contredit la définition de d. □

Lemme 5.1.31. *À tout instant, pour tout $\ell \in \mathbb{N}$, l'ensemble des lignes $L_{\ell'}$ (respectivement colonnes $C_{\ell'}$) avec $\ell' \geq \ell$ est tel que au plus $\mathcal{U}_\ell$ de ces lignes (colonnes) sont* utilisées *et au plus $\mathcal{C}_\ell$ processus sont* contenus *dans une de ces lignes (colonnes). On a alors $\mathcal{C}_\ell + \mathcal{U}_\ell \leq 2n - \ell$.*

Démonstration. Pour $\ell = 0$, la propriété est vraie, car au plus n processus existent dans le système, et donc au plus n d'entre eux sont contenus par une ligne (colonne). De plus, chaque processus est occupé dans au plus 1 diviseur, donc au plus n diviseurs sont occupés, et c'est donc aussi le cas pour les lignes (colonnes), d'où $\mathcal{U}_0 \leq n$.

De plus, initialement, aucun processus n'est contenu dans la grille et aucune ligne (colonne) n'est occupée, donc le lemme est vrai. Considérons maintenant une exécution de cet algorithme de renommage, et supposons par induction que jusqu'à un instant t donné, ce lemme soit vrai. À t, les évènements susceptibles de rendre ce lemme faux (en augmentant le nombre de processus contenus dans une ligne/colonne ou le nombre de lignes/colonnes utilisées) sont l'entrée d'un processus dans la grille (c'est-à-dire dans le diviseur $\langle 0, 0\rangle$), ou le déplacement d'un processus d'un diviseur vers un autre (après avoir obtenu *droite* ou *bas*), ou la terminaison de *Relache* après un retour de *stop* correspondant par un diviseur. Dans le premier cas, cela ne modifie pas les lignes $L_{\ell'}$ (colonnes $C_{\ell'}$), $\ell' \geq 1$, et le lemme reste donc vrai pour $\ell \geq 1$. De plus, le lemme est vrai pour $\ell = 0$, comme cela a été expliqué.

Dans le cas où un processus p_i passe d'une ligne L_ℓ (colonne C_ℓ) à une ligne $L_{\ell+1}$ (colonne $C_{\ell+1}$), alors immédiatemment avant que p_i ne reçoive *bas* (respectivement *droite*), et donc entre le moment où L_ℓ (C_ℓ) contient p_i et celui où $L_{\ell+1}$ ($C_{\ell+1}$) contient p_i, le lemme est supposé vrai et donc $\mathcal{U}_\ell + \mathcal{C}_\ell \leq 2n - \ell$. Si à ce moment là, un autre processus p_j est aussi contenu dans la ligne L_ℓ (colonne C_ℓ), alors cela signifie que $\mathcal{C}_\ell - 2 \geq \mathcal{C}_{\ell+1}$, car au moins 2 processus des lignes $L_{\ell'}$ (colonnes $C_{\ell'}$), $\ell' \geq \ell$, sont contenus dans la ligne L_ℓ (colonne C_ℓ). D'où $\mathcal{U}_{\ell+1} + \mathcal{C}_{\ell+1} \leq \mathcal{U}_\ell + \mathcal{C}_\ell - 2 \leq 2n - \ell - 2$. Dans le cas contraire, on a la ligne L_ℓ (colonne C_ℓ) qui arrive à la fin d'une période occupée, et donc d'après le lemme précédent, au moins un diviseur a retourné *stop* au cours de cette période. Donc, cette ligne (colonne) est *utilisée* (car aucun autre processus n'est occupé), ce qui signifie que $\mathcal{C}_\ell - 1 \geq \mathcal{C}_{\ell+1}$ (car p_i est contenu dans la ligne L_ℓ/colonne C_ℓ) et $\mathcal{U}_\ell - 1 \geq \mathcal{U}_{\ell+1}$. D'où on a encore $\mathcal{U}_{\ell+1} + \mathcal{C}_{\ell+1} \leq 2n - \ell - 2$. Cela signifie dans les deux cas que, une fois que p_i change de ligne (colonne), $\mathcal{U}_{\ell+1} + \mathcal{C}_{\ell+1} \leq 2n - \ell - 1$, car le nombre de processus contenus dans ces lignes (colonnes) augmente de 1 (grâce à p_i), mais le nombre de lignes (colonnes) utilisées ne change pas.

Enfin, lorsqu'un processus p_i appelle et termine une opération *Relache* sur un diviseur de la ligne L_ℓ (colonne C_ℓ), alors p_i n'est plus contenu dans ce diviseur, et cela compense l'augmentation possible du nombre de lignes (colonnes) utilisées. Plus précisément, si avant cet événement on avait $\mathcal{C}_\ell^{avant} + \mathcal{U}_\ell^{avant} \leq 2n - \ell$, on a après $\mathcal{C}_\ell^{apres} = \mathcal{C}_\ell^{avant} - 1$ et $\mathcal{U}_\ell^{apres} \leq \mathcal{U}_\ell^{avant} + 1$, d'où $\mathcal{C}_\ell^{apres} + \mathcal{U}_\ell^{apres} \leq 2n - \ell$. □

On peut déduire de ce lemme que, pour $\ell = 2n$, l'ensemble des lignes $L_{\ell'}$ (colonnes $C_{\ell'}$) avec $\ell' \geq \ell$ contient au plus $2n - 2n = 0$ processus, ce qui signifie que chaque processus ne peut pas atteindre un diviseur $\langle a, b \rangle$ avec $a \geq 2n$ ou $b \geq 2n$, et doit donc obtenir *stop* avant. D'où une valeur de retour x associée de manière unique à un couple $\langle a, b \rangle$ avec $a < 2n$ et $b < 2n$.

On a donc :

Théorème 5.1.32. *L'algorithme 5.5 est un algorithme de M-renommage réutilisable, avec $M = (2n)^2$.*

5.2 Extension du renommage et des diviseurs à-un-coup

Les résultats présentés dans cette partie et la suivante ont été exposés dans l'article [30], et considèrent le modèle asynchrone par envoi de messages avec $f \geq n/2$ pannes possibles.

5.2.1 Lien avec les résultats du chapitre précédent

Comme lors du chapitre 4, on va ici, ainsi que dans la partie 5.3, s'intéresser à l'extension d'objets pouvant être implémentés lorsque $f < n/2$ mais pas quand $f \geq n/2$ (comme cela sera démontré en temps voulu). Le problème du renommage sera lui aussi étendu, et résolu en utilisant une extension des diviseurs.

Il pourrait donc sembler naturel de réutiliser les résultats précédents que sont les bancs de registres χ-colorés, ou les α-registres. Cependant, ce ne sera pas le cas, pour plusieurs raisons. D'une part, les diviseurs ont une implémentation très simple, ne nécessitant pas toutes les propriétés des registres classiques, et peuvent donc profiter sans réelle perte d'une implémentation par envoi de messages. D'autre part, il est possible d'implémenter des k-diviseurs (introduits dans la partie suivante) avec $k = \lfloor n/(n-f) \rfloor$, et de même pour le renommage k-redondant. Or, la borne χ ou α des extensions de registres correspondantes

sont de l'ordre de $2f-n+2$, et ces deux bornes vérifient $2f-n+2 > \lfloor n/(n-f) \rfloor$ lorsque $n/2 < f < n-1$. Ainsi, le résultat obtenu pourrait être considéré comme *moins bon* ou *non-optimal* en utilisant ces autres registres pour implémenter des extensions de diviseurs.

Il peut cependant être intéressant de noter qu'il est possible d'implémenter des χ-diviseurs très simplement à l'aide de bancs de registres χ-colorés, en dupliquant χ fois l'algorithme d'implémentation des diviseurs classiques. Mais ce résultat n'étant pas optimal, il n'a donc pas été développé.

5.2.2 Définitions

Le renommage *à-un-coup* classique (défini partie 5.1.1) peut être résolu dans le modèle asynchrone par envoi de messages si et seulement si $f < n/2$, d'après le lemme 5.2.1.

Pour cette raison, nous avons défini une version étendue de ce problème, appelée renommage k-redondant à-un-coup, pour laquelle jusqu'à k processus peuvent obtenir un même *nom* au lieu d'au plus 1. Ce problème est correctement utilisé si :

1. Initialement, chaque processus p_i a un identifiant unique id_i.
2. Chaque processus ne peut participer qu'une seule fois au renommage.

Et vérifie les propriétés suivantes :

1. Chaque processus correct p_i doit retourner une valeur x_i en un temps fini (après avoir retourné une valeur, p_i peut continuer d'exécuter un algorithme, mais il ne peut plus modifier sa valeur de retour).
2. Si $k+1$ processus distincts $p_{i_0}, \ldots, p_{i_k}$ retournent $k+1$ valeurs $x_{i_0}, \ldots, x_{i_k}$, alors au moins 2 de ces valeurs sont distinctes, c'est-à-dire $\exists a, b \in [0,k] : x_{i_a} \neq x_{i_b}$.

En particulier, on peut remarquer que si $k = 1$, alors cette définition correspond exactement au renommage à-un-coup classique. La dernière propriété peut être reformulée en : toute valeur x peut être retournée par au plus k processus.

On pourra encore parler de renommage dans un espace de nom de taille M, même dans le cas k-redondant, et on pourra le noter (M,k)-renommage à-un-coup. De plus, on dira qu'un problème de renommage est *non-trivial* s'il s'agit d'un problème de M-renommage avec un certain $M \in \mathbb{N}$ qui peut dépendre de n et f mais pas de la taille des identifiants.

On démontre alors le lemme suivant, qui implique entre autre l'impossibilité de résoudre le renommage classique (ou 1-renommage) non-trivial lorsque $f \geq n/2$:

Lemme 5.2.1. *Pour un système distribué avec n et f fixé, il est impossible de résoudre un problème de renommage k-redondant non-trivial si $k < \lfloor n/(n-f) \rfloor$.*

Démonstration. Supposons qu'il existe un algorithme capable de résoudre le (M,k)-renommage avec M qui dépend uniquement de n et f (donc fixé pour un système donné), et $k < \lfloor n/(n-f) \rfloor$. Considérons une famille d'exécutions définies comme suit :

Les $n-f$ premiers processus $p_1, \ldots, p_{n-f}$ sont corrects, exécutent l'algorithme de (M,k)-renommage, et obtiennent chacun une valeur $x_1, \ldots, x_{n-f}$. Les f processus restants tombent en panne dès le début de l'algorithme. Pour un n-uplet d'identifiants (tous distincts deux à deux) fixé $\langle id_1, \ldots, id_n \rangle$, il peut exister plusieurs exécutions répondant aux critères précédents, mais considérons-en une en particulier, notée $E_{id_1,\ldots,id_n}$. Comme les processus $p_{n-f+1}, \ldots, p_n$ tombent en panne dès le début de l'exécution, celle-ci ne dépend pas de leurs identifiants (qui ne peuvent être connus d'aucun processus correct), et on pourra donc noter cette exécution $E_{id_1,\ldots,id_{n-f}}$ Pour cette exécution, les valeurs de retour $x_1, \ldots, x_{n-f}$ sont donc fixées, et on notera $X_{id_1,\ldots,id_{n-f}}$ l'ensemble $\{x_1, \ldots, x_{n-f}\}$. Comme il existe une infinité d'identifiants possibles, il existe une infinité d'ensembles $I = \{id_1, \ldots, id_{n-f}\}$ deux à deux disjoints, et à chacun de ces ensemble d'identifiants correspond un ensemble de valeurs X_I. Or, ces ensembles de valeurs contiennent des valeurs

dans l'ensemble fini $[1, M]$. Donc, il existe au moins une valeur x telle qu'une infinité de X_I contienne ce x, avec les ensemble d'identifiants correspondants deux à deux disjoints. En particulier, il existe une infinité d'exécution E_I telles qu'un processus obtienne x via l'algorithme de renommage.

Or, une exécution dans laquelle plusieurs groupes de $n - f$ processus sont corrects, mais tels que les messages d'un groupe vers un autre sont arbitrairement ralentis, est indistinguable (du point de vue des processus) d'une exécution durant laquelle un seul de ces groupes est correct et les autres processus sont tombés en panne. Donc, pour deux ensembles disjoints de $n - f$ identifiants I, I', l'exécution $E_{I,I'}$ durant laquelle les deux groupes de $n - f$ processus correspondants sont corrects et les messages de l'un vers l'autre sont arbitrairement ralentis est indistinguable de E_I et $E_{I'}$ respectivement. En particulier, si X_I et $X_{I'}$ contiennent tous deux x, alors lors de l'exécution $E_{I,I'}$, au moins 2 processus obtiennent la valeur x. De même, on peut itérer ce raisonnement pour une exécution $E_{I_1,\ldots,I_l}$ (avec $l = \lfloor n/(n-f) \rfloor$, car il est possible d'avoir l groupes de $n - f$ processus deux à deux disjoints), et on aura donc au moins $l > k$ processus obtenant la même valeur x. □

Il est possible d'implémenter des diviseurs *à-un-coup* classiques à l'aide de registres atomiques. Or, de tels registres peuvent être implémentés dans notre modèle par envoi de message si une stricte minorité des processus est susceptible de tomber en panne ($f < n/2$).

Malheureusement, il est impossible de simuler de tels diviseurs dans ce modèle avec une majorité de pannes possibles ($f \geq n/2$). C'est pourquoi, nous avons défini une extension de ces objets, appelés k-diviseurs à-un-coup, tels qu'au plus k processus peuvent obtenir *stop* au lieu d'au plus 1. Ces objets doivent être utilisés en ayant :

1. Chaque processus ne peut invoquer l'opération Divise sur cet objet qu'au plus *une* fois.

Et vérifient alors :

1. Tout appel à Divise qui termine retourne une valeur parmi $\{bas, droite, stop\}$.
2. Si un processus correct invoque Divise, l'appel termine.
3. Au cours de toute exécution, au plus k appels à Divise retournent *stop*.
4. Si au cours d'une exécution, p processus invoquent Divise sur cet objet, au plus $p-1$ de ces appels retournent *bas*, et au plus $p-1$ de ces appels retournent *droite*.

On peut constater que seule la troisième propriété est modifiée par rapport à la version classique des diviseurs à-un-coup. Ainsi, pour $k = 1$, un 1-diviseur à-un-coup est *exactement* un diviseur à-un-coup classique.

On montre alors la propriété suivante, qui implique l'impossibilité d'implémenter des diviseurs habituels lorsque $f \geq n/2$, et justifie donc ces nouveaux objets :

Lemme 5.2.2. *Il est impossible de simuler un k-diviseur* à-un-coup *dans un modèle asynchrone par envoi de message avec $f \geq n/2$ processus susceptibles de tomber en panne si $k < \lfloor n/(n-f) \rfloor$.*

Démonstration. Supposons qu'il existe un algorithme permettant d'implémenter un tel k-diviseur. On note $l = \lfloor n/(n-f) \rfloor > k$, et $Q_1, \ldots, Q_l$ les ensembles (disjoints) de processus définis par $Q_i = \{p_{1+(i-1)*(n-f)}, \ldots, p_{i*(n-f)}\}$. En particulier, on a $|Q_i| = n - f$.

Pour un certain $i \in [1, l]$, on considère l'exécution $\mathcal{E}_i$ durant laquelle tous les processus $\notin Q_i$ tombent en panne dès le début, alors que les processus de Q_i sont tous corrects. Ensuite, le processus $p_{i*(n-f)}$ (seul) invoque *Divise* sur le k-diviseur, et donc par définition, il obtient un réponse en temps fini, et cette réponse est nécessairement *stop*.

Considérons désormais une exécution $\mathcal{E}$ durant laquelle les processus $p_j, j > l * (n - f)$ tombent immédiatement en panne (s'ils existent), et chaque message envoyé par un processus $p \in Q_i$ à un processus $p' \notin Q_i$ est arbitrairement ralenti. Durant cette exécution, pour chaque processus de Q_i, la situation est indistinguable de celle de $\mathcal{E}_i$ durant laquelle tous les processus $\notin Q_i$ sont tombés en panne. Ensuite, lors de $\mathcal{E}$, chaque processus $p_{i*(n-f)} \forall i \in [1, l]$ invoque l'opération *Divise*. Comme la situation est indistinguable de chaque exécution $\mathcal{E}_i$, chacun de ces appels retourne *stop*. Donc, $l > k$ appels ont retourné *stop*, ce qui est impossible par définition du k-diviseur. Cette contradiction démontre ce lemme. □

Comme leur version habituelle, ces k-diviseurs peuvent être utilisés pour le renommage (partie 5.2.4), et pourraient aussi être utiles pour d'autres problèmes de par leurs propriétés intéressantes et leur implémentation relativement simple. En plus de leur utilisation explicite présentée plus tard, ces objets sont donc intéressantes en soi, notamment parce qu'il s'agit d'une simple extension intuitive des diviseurs classiques.

5.2.3 Implémentation des k-diviseurs à-un-coup

L'algorithme 5.6 est une implémentation d'un k-diviseur à-un-coup avec $k = \lfloor n/(n - f) \rfloor$. Il s'inspire de l'algorithme issu de [78] présenté dans la partie 5.1.2. Cependant, à cause de l'impossibilité de simuler des registres atomiques, certaines modifications sont nécessaires afin de s'approcher du comportement des registres. De plus, afin de réduire le k à $\lfloor n/(n-f) \rfloor$, plusieurs *Portes* sont ajoutées. Pour simuler cela, et parce que les portes sont successives, $porte_i$ contient un entier qui correspond au nombre de portes fermées, ou (de manière équivalente) au numéro de la dernière porte fermée. Enfin, la variable nom_i (basée sur le registre Nom), contient un identifiant qui n'est pas nécessairement le *dernier* écrit, mais plutôt celui le *plus grand* (les identifiants étant des entiers, donc comparables).

Le principe de l'algorithme est d'*écrire* son identifiant id_i, puis d'effectuer jusqu'à $N = \lfloor n/(n - f) \rfloor + 1$ rondes chacune composée de :

- Le processus vérifie l'état de la porte suivante
- Si elle est ouverte, le processus la ferme et vérifie l'état de Nom, sinon il retourne *droite*
- Si l'identifiant dans Nom n'est pas le sien, le processus retourne *bas*.

Enfin, un processus qui termine les N rondes successives retourne *stop*.

Cet algorithme peut être représenté par un schéma 5.8 similaire à celui utilisé pour les diviseurs habituels, mais les *portes* et le *registre Nom* ne représentent ici plus des registres, et ne vérifient donc pas toutes les propriétés souhaitées, raison pour laquelle il est possible que k processus obtiennent *stop*.

Algorithme 5.6 Implémentation de k-diviseur à-un-coup. (code exécuté par p_i)

```
1:  Initialisation
2:      portes_i ← 0 ; nom_i ← −∞ ;
3:  Fonction DIVISE(id_i)
4:      nom_i ← max(nom_i, id_i) ;
5:      pour r_i de 1 à N faire                          ▷ N = ⌊n/(n−f)⌋ + 1
6:          pour chaque processus p faire
7:              envoyer (Test, r_i) à p ;
8:          attendre de recevoir n − f messages (RepTest, ∗, r_i) ;
9:          portes_i ← max({po : (RepTest, po, r_i) a été reçu} ∪ {portes_i}) ;
10:         si (portes_i ≥ r_i) alors retourner droite ;
11:         pour chaque processus p faire
```

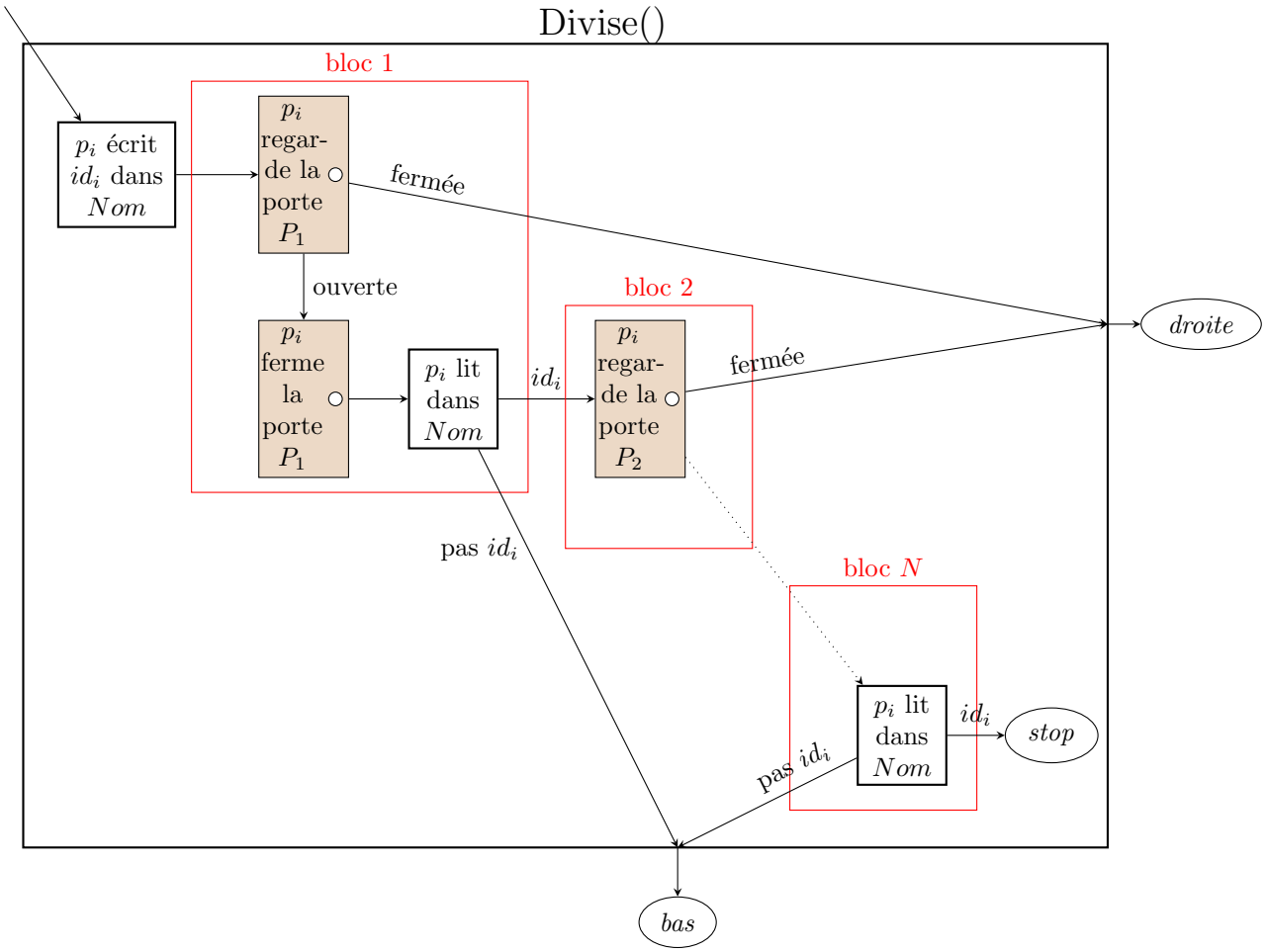

FIGURE 5.8 – Implémentation d'un k-diviseur à-un-coup

12: **envoyer** (Id, $nom_i, portes_i, r_i$) **à** p ;
13: **attendre** de recevoir $n - f$ messages (RepId,$*, *, r_i$) ;
14: $nom_i \leftarrow \max(\{id : (\text{RepId}, id, *, r_i)$ a été reçu$\} \cup \{nom_i\})$;
15: $portes_i \leftarrow \max(\{po : (\text{RepId}, *, po, r_i)$ a été reçu$\} \cup \{portes_i\})$;
16: **si** $nom_i \neq id_i$ **alors retourner** bas ;
17: **retourner** $stop$;
18: **Quand** un message m **est reçu** du processus p_j
19: **selon** la forme de m
20: **cas** $m = (\text{Test}, r)$ **faire**
21: **envoyer** (RepTest, $portes_i, r$) **à** p_j ;
22: **cas** $m = (\text{Id}, id, po, r)$ **faire**
23: $nom_i \leftarrow \max(id, nom_i)$; $portes_i \leftarrow \max(po, portes_i)$;
24: **envoyer** (RepId, $nom_i, portes_i, r$) **à** p_j ;

On supposera que la fonction *Divise* ainsi implémentée est correctement utilisée, c'est-à-dire que chaque processus ne peut l'invoquer qu'au plus *une* fois. Les lemmes suivants démontrent alors que l'algorithme 5.6 implémente bien un k-diviseur à-un-coup avec $k = \lfloor n/(n-f) \rfloor$.

Observation 5.2.3. *Tout appel à Divise qui termine retourne une valeur parmi* $\{bas, droite, stop\}$.

Lemme 5.2.4. *Tout appel à Divise par un processus correct termine.*

Démonstration. Soit p_i un processus correct. Chaque processus correct finit par recevoir les messages (Test, r) et (Id, $*, *, r$) envoyés par p_i et répondre en envoyant (RepTest, $*, r$)

et (RepId, $*, *, r$) lignes 21 et 24. Donc, comme au moins $n - f$ processus sont corrects, p_i termine nécessairement les lignes 8 et 13. Comme le nombre de rondes est borné (au plus N), on peut en déduire que l'appel de *Divise* par p_i termine. □

On peut remarquer qu'à chaque fois que $portes_i$ ou nom_i prend une nouvelle valeur, cette valeur est au moins aussi grande que la précédente (lignes 4, 9, 14,15, 23), d'où :

Observation 5.2.5. *Pour chaque processus p_i, les valeurs successives contenues dans les variables $portes_i$ et nom_i respectivement forment une suite croissante.*

Lemme 5.2.6. *Soit p le nombre de processus qui invoquent Divise au cours d'une exécution. Au plus $p - 1$ de ces appels retournent bas.*

Démonstration. Si tous les appels ne terminent pas, alors ce lemme est vrai, on supposera donc que tous les appels terminent. Soit p_m (d'identifiant id_m) le processus ayant le plus grand identifiant parmi les p processus qui invoquent *Divise*. Au début de cette invocation, nom_m reçoit id_m (ligne 4). Or, les valeurs successives de nom_m sont croissantes et ne peuvent contenir que des identifiants de processus ayant appelé *Divise* (lignes 14 et 23). Donc, après que p_m ait invoqué *Divise*, $nom_m = id_m$ et reste inchangé (par définition de id_m). Donc, p_m ne peut pas retourner *bas*, car pour cela il faudrait vérifier $nom_m \neq id_m$ (ligne 16). □

Lemme 5.2.7. *Soit p le nombre de processus qui invoquent Divise au cours d'une exécution. Au plus $p - 1$ de ces appels retournent droite.*

Démonstration. Supposons par contradiction que les p appels retournent tous *droite*. Pour un processus p_i, on notera alors $rd_i \in [1, N]$ la valeur de r_i lorsque p_i retourne *droite* (ligne 10). Parmi les p processus que l'on considère, soit p_m un processus ayant le rd_m maximal (il est possible que plusieurs processus aient le même rd_i). D'après l'algorithme, lors de la ronde rd_m, p_m reçoit un message (RepTest, $po, *$) avec $po \geq rd_m$ d'un processus p_i. Ce processus p_i vérifiait $portes_i = po$ lorsqu'il a envoyé ce message. Or, la valeur de $portes_i$ est remplacée par po lorsque p_i exécute la ronde po sans retourner *droite* (ligne 9) ou quand il reçoit un message (RepId, $*, po, *$) ou (Id, $*, po, *$) (lignes 15 et 23 respectivement). Dans le premier cas, p_i ne peut pas retourner *droite*, car s'il le faisait ce serait lors d'une ronde $r > po \geq rd_m$. Dans le second cas, le même raisonnement s'applique au processus $p_{i1} \notin \{p_i, p_m\}$ qui a envoyé le message correspondant (RepId, $*, po, *$) ou (Id, $*, po, *$). Récursivement, on peut en déduire que soit p_{i1} ne retourne pas *droite*, soit il reçoit un message similaire de la part d'un autre processus p_{i2}, et ainsi de suite. Comme il n'y a qu'un nombre fini de processus, il y a nécessairement un processus qui a modifié son $portes_\ell$ en po à la ligne 9, et ce processus ne peut pas retourner *droite*. □

Lemme 5.2.8. *Au plus $\lfloor \frac{n}{n-f} \rfloor$ processus retournent* stop.

Démonstration. Soit p_i un processus qui termine la r-ième ronde (lignes 5–16) sans retourner *bas* ou *droite*, pour un certain $r \in [1, N]$. Autrement dit, p_i commence ensuite la ronde $r+1$ ou, si $r = N$, retourne *stop*. Soit $M(r, p_i)$ l'ensemble de $n-f$ messages (RepId, $*, *, r$) reçus par p_i lors de la ronde r, et soit $Q(r, p_i)$ l'ensemble des $n - f$ processus ayant envoyé ces messages. Enfin, on notera $portes_i^r$ et nom_i^r les valeurs des variables $portes_i$ et nom_i (respectivement) à la fin de la ronde r. On peut noter que, comme p_i ne retourne pas *bas* lors de cette ronde, $nom_i^r = id_i$ (ligne 16).

On démontre désormais une propriété intermédiaire : Soient p_i et $p_{i'}$ deux processus qui retournent tous deux *stop*, et $1 \leq r < r' \leq N$. Si $id_i > id_{i'}$, alors $Q(r, p_i) \cap Q(r', p_{i'}) = \emptyset$.

Pour démontrer cette propriété, on suppose par contradiction que $Q(r, p_i) \cap Q(r', p_{i'}) \neq \emptyset$ et $p_k \in Q(r, p_i) \cap Q(r', p_{i'})$. Par définition de Q, p_k envoie un message $m =$ (RepId,

$\ell, po, r)$ et $m' = (\text{RepId}, \ell', po', r')$ à p_i et $p_{i'}$ respectivement. On peut observer que $\ell = id_i$ et $\ell' = id_{i'}$. En effet, avant d'envoyer ces réponses, p_k a reçu des messages de la forme (Id, $\ell_i, po_i, r)$ vérifiant $\ell_i \geq id_i$ de par la croissance des valeurs contenues dans nom_i, et de même p_k doit alors vérifier $nom_k \geq \ell_i$ (ligne 23) et $\ell = nom_k$. De plus, si p_i obtient une valeur dans nom_i qui est plus grande que id_i, alors il retourne *bas*. Donc, $id_i \leq \ell_i \leq \ell \leq id_i$, ce qui montre bien que les messages envoyés par p_k sont $m = (\text{RepId}, id_i, po, r)$ et $m' = (\text{RepId}, id_{i'}, po', r')$.

Comme $id_i > id_{i'}$ et que les valeurs stockées dans nom_k sont croissantes, on peut en déduire que le message m' a été envoyé avant le message m (qui contiennent tous deux la valeur de nom_k au moment de l'envoi). Le message m' répond à un message (Id, $id_{i'}, po_{i'}, r')$ envoyé par $p_{i'}$ lors de la ronde r'. $po_{i'}$ est la valeur contenue dans $portes_{i'}$ lors de l'envoi de ce message, et vérifie donc $po_{i'} \geq r'$ (ligne 12). Après la réception de ce message, $portes_k \geq po_{i'} \geq r'$ (ligne 23). Comme m est envoyé après m' et que les valeurs successives de $portes_k$ sont croissantes, le po envoyé dans le message m était contenu dans $portes_k$ et vérifie donc $po \geq r'$.

Lorsque p_i reçoit ce message m, lors de la ronde r, la valeur de $portes_i$ devient plus grande que po si ce n'était pas déjà le cas (ligne 15). Donc, $portes_i^r \geq r' > r$. Comme $r < N$ et p_i doit retourner *stop*, il exécute la ronde $r+1$. Mais comme la valeur de $portes_i$ est croissante, $portes_i \geq r+1$ lors de cette ronde, cela signifie que p_i retourne *droite* (ligne 10). Cette contradiction démontre la propriété souhaitée.

Pour $r \in [1, N]$, soit E_r l'ensemble des processus qui terminent la ronde r sans retourner *bas* ou *droite*. On peut noter que les processus retournant *stop* appartiennent à E_N. On souhaite donc montrer que $|E_N| \leq \lfloor \frac{n}{n-f} \rfloor$.

En notant $m = |E_N|$, soit $E_N = \{p_1, \ldots, p_m\}$, avec $id_1 > \ldots > id_m$ (quitte à réordonner les processus), et supposons par contradiction que $m \geq \lfloor \frac{n}{n-f} \rfloor + 1 = N$. Pour tout i, j vérifiant $1 \leq i < j \leq N$, on a d'après la propriété précédente, $Q(i, p_i) \cap Q(j, p_j) = \emptyset$ (car $id_i > id_j$). Par définition, chacun de ces $Q(i, p_i)$ est un ensemble de $n - f$ processus. Donc, $|\bigcup_{1 \leq i \leq N} Q(i, p_i)| = N * (n - f) > n$, ce qui est une contradiction car la système comprend n processus. □

On peut donc déduire de ces lemmes :

Théorème 5.2.9. *L'algorithme 5.6 implémente un $\lfloor n/(n-f) \rfloor$-diviseur à-un-coup.*

5.2.4 k-renommage à-un-coup

On s'intéresse maintenant à un algorithme permettant de résoudre le (M, k)-renommage pour un certain M et k.

Considérons l'algorithme 5.1 de M-renommage (avec $M = n + f$) classique présenté en partie 5.1.1 et originaire de l'article [17]. Il s'avère que cet algorithme permet aussi de résoudre le (M, k)-renommage avec $k = \lfloor n/(n-f) \rfloor$, et on peut remarquer que le $(M, 1)$-renommage est exactement du M-renommage, et que $k = 1$ lorsque $f < n/2$.

Les lemmes suivants permettent de démontrer ce résultat.

Le lemme 5.1.1 peut être généralisé lorsque $f \geq n/2$.

Lemme 5.2.10. *Pour tout algorithme A localement correct, si on considère $k + 1 = \lfloor n/(n-f) \rfloor + 1$ tableaux $V^0, \ldots, V^k$ qui sont* stables *à certains moments lors d'une exécution de A, alors au moins deux de ces tableaux V^y et V^z $(y \neq z)$ sont comparables : $V^y \leq V^z$.*

Démonstration. Lors de cette exécution, chaque tableau V^x, $x \in [0, k]$, est stable. Cela signifie que chacun de ces tableaux a été, à un moment, dans la variable V_j d'un processus

p_j, et ce pour au moins $n - f$ processus différents. On note P_x l'ensemble des processus ayant contenu V^x dans leur variable locale V_j à un moment lors de cette exécution. Comme $|P_x| \geq n - f \ \forall x \in [0, k]$, cela signifie que $|\bigcup_{x\in[0,k]} P_x| \leq n < \sum_{x\in[0,k]} |P_x|$, et donc il existe $y \neq z$ tels que $P_y \cap P_z \neq \emptyset$. Soit $p_i \in P_y \cap P_z$. A un moment de l'exécution, V_i contient V^y, et à un autre moment V_i contient V^z. Comme A est localement correct, alors $V^y \leq V^z$ ou $V^z \leq V^y$. □

L'algorithme en question est bien localement correct, comme démontré dans le lemme 5.1.2. De plus, les lemmes 5.1.3 et 5.1.4 sont encore vrais en présence d'une majorité de pannes.

On remplace alors le lemme 5.1.5, uniquement valable lorsque $f < n/2$, par le lemme suivant :

Lemme 5.2.11. *Pas plus de $k = \lfloor n/(n-f) \rfloor$ processus différents ne peuvent retourner la même valeur x lors d'une exécution de l'algorithme 5.1.*

Démonstration. Supposons par contradiction qu'au moins $k + 1$ processus retournent la même valeur x. Soient $V^0, \ldots, V^k$ les tableaux stables correspondants aux retours de ces processus (c'est-à-dire la valeur contenue dans V_i lorsque p_i termine son appel à *Renomme*). On sait qu'il existe $y \neq z$ tels que $V^y \leq V^z$. Soient p^y et p^z les processus (distincts) correspondants à ces tableaux stables d'identifiants id^y et id^z. D'après les lemmes 5.1.3 et 5.1.4, on sait que $V^y[id^y].x = x$, $V^z[id^z].x = x$, et $V^z[id^y] = V^y[id^y]$. Cela signifie donc que, lorsque p^z a terminé, au moins deux cases $V^z[id^z]$ et $V^z[id^y]$ contiennent x, mais dans ces conditions, p^z ne peut pas retourner x (ligne 21). □

Comme les valeurs de retour sont dans l'ensemble $[0, n + f - 1]$, il ne reste plus qu'à montrer que chaque appel par un processus correct termine, et cet algorithme sera donc un algorithme de $(n + f, \lfloor n/(n - f) \rfloor)$-renommage.

Les preuves des lemmes 5.1.6 à 5.1.9 n'utilisent pas l'hypothèse que $f < n/2$. Donc, ils sont toujours vrais lorsque $f \geq n/2$, ce qui démontre que tout processus correct finit par retourner une valeur.

On a donc :

Théorème 5.2.12. *L'algorithme 5.1, présenté en partie 5.1.1, est un algorithme de $(n + f, \lfloor n/(n - f) \rfloor)$-renommage à-un-coup.*

Une autre possibilité d'algorithme de (M, k)-renommage est d'utiliser une grille de k-diviseurs comme la grille de diviseurs utilisés pour du renommage classique, présentée via l'algorithme 5.3 en partie 5.1.3.

En effet, la démonstration que chaque processus obtient une valeur en temps fini, et que les valeurs ainsi obtenues sont dans l'ensemble $[0, n(n + 1)/2 - 1]$ utilise la propriété de terminaison des diviseurs, et celles limitant les retours de *droite* et *bas* à au plus $p - 1$ sur p processus appelant *Divise*. Ces propriétés sont identiques pour les k-diviseurs, donc il ne reste qu'à démontrer que chaque valeur x est retournée par au plus k processus.

Lemme 5.2.13. *Lors de l'algorithme 5.3 présenté en partie 5.1.3, si les diviseurs sont remplacés par des k-diviseurs, alors pour toute valeur x, au plus k processus peuvent retourner x.*

Démonstration. Si un processus p_i retourne une valeur x_i, cette valeur est associée de manière unique à un $\langle a_i, b_i \rangle$ correspondant à un unique k-diviseur D. Et cela ne peut avoir lieu que si p_i a obtenu *stop* de D. Donc, si au moins $k + 1$ processus retournent la même valeur x, cela signifie que chacun de ces processus a obtenu *stop* du même k-diviseur D, ce qui contredit la propriété de ces objets. □

On a donc :

Théorème 5.2.14. *L'algorithme 5.3 de M-renommage à-un-coup ($M = n(n+1)/2$) utilisant des diviseurs est aussi un algorithme de (M,k)-renommage si les diviseurs sont remplacés par des k-diviseurs à-un-coup.*

Plus généralement, tout algorithme de M-renommage à-un-coup utilisant des diviseurs à-un-coup comme seul moyen de communication peut être traduit en algorithme de (M,k)-renommage à-un-coup en remplaçant les diviseurs par des k-diviseurs. C'est par exemple le cas de la grille de diviseurs à-un-coup présentée dans [15]

5.3 Extension du renommage et des diviseurs réutilisables

5.3.1 Définitions

Le renommage *réutilisable* classique (défini partie 5.1.5) peut être résolu (non-trivialement) dans le modèle asynchrone par envoi de messages si et seulement si $f < n/2$. La seule différence avec le renommage réutilisable est qu'ici un nom est capturé par au plus k processus à tout instant, au lieu d'au plus 1.

Pour cette raison, nous avons défini une version étendue de ce problème, appelée renommage k-redondant réutilisable, qui est correctement utilisée si :

1. Initialement, chaque processus p_i a un identifiant unique id_i.
2. Les processus peuvent chercher à obtenir une valeur et libérer une valeur précédemment obtenue, et ce autant de fois que souhaité.
3. Un processus ne peut chercher à obtenir une valeur que s'il a libéré la dernière valeur qu'il a obtenue.
4. Un processus ne peut libérer une valeur que s'il l'a précédemment obtenue et pas encore libérée depuis.

Et vérifie les propriétés :

1. Chaque processus correct p_i qui désire obtenir une valeur x_i doit l'obtenir en temps fini.
2. Chaque processus correct p_i qui désire libérer une valeur doit le faire en temps fini.
3. Si à un instant t, $k+1$ processus distincts $p_{i_0}, \ldots, p_{i_k}$ ont obtenu $k+1$ valeurs $x_{i_0}, \ldots, x_{i_k}$ et ne les ont pas encore libérées, alors deux de ces valeurs sont différentes : $\exists a, b \in [0,k] : x_{i_a} \neq x_{i_b}$.

La dernière propriétés peut être reformulée en : À tout instant t, toute valeur x peut être capturée par au plus k processus.

Encore une fois, on parlera de (M,k)-renommage réutilisable pour désigner le M-renommage k-redondant réutilisable. De plus, on peut démontrer que :

Lemme 5.3.1. *Pour un système distribué avec n et f fixé, il est impossible de résoudre un problème de renommage k-redondant réutilisable non-trivial si $k < \lfloor n/(n-f) \rfloor$.*

Démonstration. La preuve est identique à celle du lemme 5.2.1, si ce n'est qu'on peut préciser que lors de chacune des exécutions considérées, aucun processus n'appelle la fonction *Relache*. □

Ce lemme démontre par ailleurs qu'il est impossible de résoudre le renommage réutilisable non-trivial si $f \geq n/2$.

De même que pour leur version *à-un-coup*, les diviseurs *réutilisables* classiques peuvent être implémentés lorsque $f < n/2$ mais pas quand $f \geq n/2$. Il est possible de définir une version étendue de ces diviseurs que sont les k-diviseurs *réutilisables*.

Pour rappel, un diviseur est considéré comme *occupé* si un appel à *Divise* sur lui n'a pas terminé, ou si un appel à *Divise* a retourné *stop* et que l'appel correspondant à *Relache* n'a pas encore été invoqué. Dans sa nouvelle version, comme pour le cas à-un-coup, le diviseur réutilisable peut être capturé (avoir retourné *stop* et pas encore appelé Relache) par au plus k processus à tout instant (au lieu d'au plus 1). Cependant, il ne s'agit ici pas de la seule différence. En effet, toutes les périodes *occupées* ne vérifient pas la propriété limitant les retours de Divise à pas tous droite ni tous bas. Les périodes occupées vérifiant cette propriété sont alors appelées *convenables*, et la première période occupée du diviseur est bien *convenable*, ce qui signifie que s'il est utilisé au plus une seule fois par chaque processus, cet objet se comporte comme un k-diviseur à-un-coup.

Un k-diviseur réutilisable doit alors vérifier :

1. Un processus ne peut appeler Relache que si son dernier appel à Divise a retourné *stop*, et qu'il n'a pas invoqué Relache depuis le retour de cette opération Divise.
2. Un processus ne peut pas appeler Divise si son dernier appel à Divise a retourné *stop* et qu'il n'a pas encore invoqué Relache depuis cette opération Divise.

Et a les propriétés :

1. Si un appel à Divise termine, il retourne une valeur dans $\{bas, droite, stop\}$
2. Si un processus correct invoque Divise ou Relache, cette opération termine.
3. À tout instant, au plus k processus ont obtenu *stop* lors d'un appel à Divise et n'ont pas appelé l'opération Relache depuis.
4. Tout intervalle de temps durant lequel chaque invocation de *Divise* retourne *bas* est fini.
5. Durant certaines périodes occupées, les appels à Divise ne retournent pas tous *droite* et pas tous *bas*. Une telle période sera appelée *convenable*.
6. Initialement, le diviseur est dans une période *libre* (définie ci-dessous), ce qui signifie que la première période occupée de ce diviseur est *convenable*.

Pour une exécution $\mathcal{E}$ et un instant t de cette exécution, on considère l'ensemble $Possible(\mathcal{E}, t)$ comme l'ensemble de toutes les exécutions possibles qui sont exactement identiques à $\mathcal{E}$ de l'instant initial à l'instant t (en particulier, $\mathcal{E} \in Possible(\mathcal{E}, t)$). À l'instant t d'une exécution $\mathcal{E}$, on dira qu'un diviseur est dans une période *libre* si il n'est pas occupé et si, pour toute exécution $\mathcal{E}' \in Possible(\mathcal{E}, t)$, la prochaine période occupée (si elle existe) de ce diviseur est *convenable*. En particulier, un diviseur qui est dans une période *libre* le reste jusqu'à sa prochaine période occupée (si elle existe).

Enfin, on peut démontrer qu'il est impossible d'implémenter un k-diviseur réutilisable avec $k < \lfloor n/(n-f) \rfloor$, de manière identique à la preuve pour les k-diviseurs à-un-coup présentée dans le lemme 5.2.2.

5.3.2 Implémentation des k-diviseurs réutilisables

L'algorithme 5.7 implémente un $\lfloor n/(n-f) \rfloor$-diviseur réutilisable.

Une fois de plus, cet algorithme est basé sur l'implémentation d'un diviseur réutilisable classique, en utilisant N portes au lieu d'une seule. De plus, les registres sont (encore une fois) remplacés par des variables locales qui sont mises à jour via des échanges de messages entre les processus. Cet algorithme peut être représenté par la figure 5.9, pour lequel les portes et le *registre* Nom ne sont pas de vrais registres.

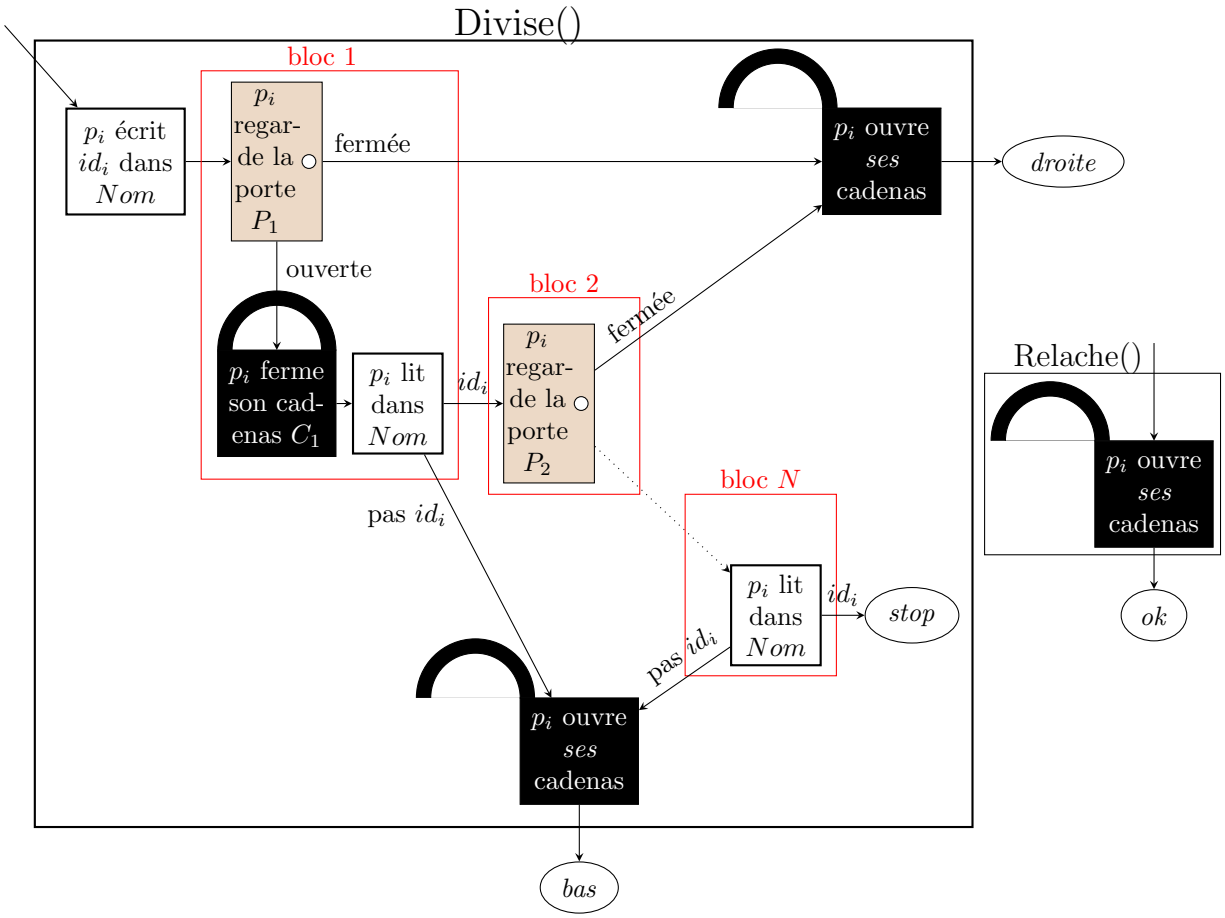

FIGURE 5.9 – Implémentation d'un k-diviseur réutilisable

Plus précisément, un registre 1WnR de porte, dont l'écrivain est p_j, est simulé d'une manière similaire à la simulation ABD (présentée en partie 3.1.2), et stocké dans les $Portes_i[id_j]$. On y associe une estampille $TPortes_i[id_j]$, permettant d'avoir des valeurs totalement ordonnées (grâce au côté mono-écrivain). Enfin, la fonction Fusionne permet de mettre à jour chaque paire $\langle TPortes_i[id], Portes_i[id]\rangle$ pour tout id à réception d'un message, en gardant la version la plus récente pour chacun des *registres* ainsi simulés.

Le système de rondes r_i est associé à une variable A_i qui compte le nombre d'appels à Divise par p_i, afin de pouvoir réinitialiser r_i sans perdre l'ordre total. Enfin, $RepTest_i$ et $RepId_i$ permettent de stocker les réponses correspondantes reçues, afin de tester ensuite la réception de $n - f$ messages suffisamment récents.

Algorithme 5.7 Implémentation de k-diviseur réutilisable. (code exécuté par p_i)

1: **Initialisation**
2: $A_i \leftarrow 0$; ▷ Compte le nombre d'appels à *Divise* par p_i
3: $RepTest_i \leftarrow \emptyset$; $RepId_i \leftarrow \emptyset$;
4: $Connus_i \leftarrow \{id_i\}$; ▷ Permet d'identifier les cases initialisées des tableaux suivants
5: $\langle TPortes_i, Portes_i\rangle \leftarrow$ 2 tableaux dynamiques d'entiers indexés par des identifiants id et dont les cases non initialisés valent 0.
6: **Fonction** DIVISE(id_i)
7: $A_i \leftarrow A_i + 1$
8: **pour** r_i **de** 1 **à** N **faire** ▷ $N = \lfloor \frac{n}{n-f} \rfloor + 1$
9: **pour chaque** processus p **faire**
10: **envoyer** (Test, $\langle T, P\rangle, C, \langle A_i, r_i\rangle$) **à** p ;
11: **attendre** que $|\{(\text{RepTest}, *, *, \langle A, r\rangle) \in RepTest_i$ tel que $\langle A, r\rangle = \langle A_i, r_i\rangle\}| \geq$

$n - f$;
12: **si** $\exists id \in Connus_i : Portes_i[id] \geq r_i$ **alors**
13: RELACHE( ) ; **retourner** *droite* ;
14: $\langle TPortes_i[id_i], Portes_i[id_i]\rangle \leftarrow \langle TPortes_i[id_i] + 1, r_i\rangle$
15: **pour chaque** processus p **faire**
16: **envoyer** (Id, $\langle TPortes_i, Portes_i\rangle, Connus_i, \langle A_i, r_i\rangle$) **à** p ;
17: **attendre** que $|\{(\text{RepId}, *, *, \langle A, r\rangle) \in RepId_i \text{ tel que } \langle A, r\rangle = \langle A_i, r_i\rangle\}| \geq n - f$;
18: **si** $\max\{id \in Connus_i$ tels que $Portes_i[id] > 0\} \neq id_i$ **alors**
19: RELACHE( ) ; **retourner** *bas* ;
20: **retourner** *stop* ;
21: **Procédure** RELACHE( )
22: $\langle TPortes_i[id], Portes_i[id]\rangle \leftarrow \langle TPortes_i[id] + 1, 0\rangle$;
23: **pour chaque** processus p **faire**
24: **envoyer** (Relache, $\langle TPortes_i, Portes_i\rangle, Connus_i$) **à** p ;
25: **Fonction** FUSIONNE($\langle T_1, P_1\rangle, \langle T_2, P_2\rangle, C$)
26: **pour chaque** $id \in C$ **faire**
27: **si** $T_2[id] > T_1[id]$ **alors** ▷ si $T_2[id] = T_1[id]$, alors $P_2[id] = P_1[id]$
28: $\langle T_1[id], P_1[id]\rangle \leftarrow \langle T_2[id], P_2[id]\rangle$;
29: **retourner** $\langle T_1, P_1\rangle$;
30: **Quand** un message $m = (type, \langle T, P\rangle, C, \langle A, r\rangle)$ **est reçu** du processus p_j
31: $Connus_i \leftarrow Connus_i \cup C$;
32: $\langle TPortes_i, Portes_i\rangle \leftarrow$ FUSIONNE($\langle TPortes_i, Portes_i\rangle, \langle T, P\rangle, Connus_i$) ;
33: **selon** *type*
34: **cas** *type* = Test **faire**
35: **envoyer** (RepTest, $\langle TPortes_i, Portes_i\rangle, Connus_i, \langle A, r\rangle$) **à** p_j
36: **cas** *type* = Id **faire**
37: **envoyer** (RepId, $\langle TPortes_i, Portes_i\rangle, Connus_i, \langle A, r\rangle$) **à** p_j ;
38: **cas** *type* = RepTest **faire**
39: $RepTest_i \leftarrow RepTest_i \cup m$;
40: **cas** *type* = RepId **faire**
41: $RepId_i \leftarrow RepId_i \cup m$;

On démontre maintenant que cet algorithme implémente bien un k-diviseur réutilisable, avec $k = \lfloor n/(n-f)\rfloor$.

Observation 5.3.2. *Si un appel à Divise termine, il retourne une valeur dans* $\{bas, droite, stop\}$.

Lemme 5.3.3. *Pour tout processus correct* p_i*, chaque appel à Divise ou à Relache termine.*

Démonstration. Chaque processus correct finit par recevoir les messages envoyés par p_i et y répondre. En particulier, le message initial contient $\langle A_i, r_i\rangle$ donc la réponse aussi. Donc, comme au moins $n - f$ processus sont corrects, chaque fois que p_i attend $n - f$ réponses, cela finit par arriver (lignes 11 et 17). Comme *Divise* se déroule en un nombre borné de rondes (au plus N), l'appel termine. Enfin, tout appel à *Relache* termine de manière évidente. □

Les valeurs des $\langle TPortes_i[id], Portes_i[id]\rangle$ sont ordonnées par la valeur de l'estampille $TPorte_i[id]$ (ligne 28). De plus, chaque $TPortes_i[id]$ est associé à un unique $Portes_i[id]$, car les deux changent toujours en même temps. D'après l'algorithme, chaque fois que

$\langle TPortes_i[id], Portes_i[id]\rangle$ est modifié, il est remplacé par une valeur plus grande (lignes 32,14), d'où :

Observation 5.3.4. *Pour chaque processus p_i et chaque identifiant id, les valeurs successives de la variable $\langle TPortes_i[id], Portes_i[id]\rangle$ forment une suite croissante.*

Lemme 5.3.5. *Considérons une exécution de cet algorithme, à un instant t, durant lequel le diviseur n'est pas occupé. Si, pour chaque processus p_i et chaque identifiant id, $Portes_i[id] = 0$, et que pour chaque message m (contenant $\langle T, P\rangle$) envoyé de p_j à p_i, pour chaque identifiant id, $P[id] = 0$ ou $T[id] < TPortes_i[id]$, alors le diviseur est dans une période libre.*

Démonstration. La première observation que l'on peut faire est que, à partir de t et jusqu'à ce que p_i invoque Divise, la valeur de $Portes_j[id_i] = 0$ pour tout processus p_j et p_i. En effet, seul p_i peut modifier le couple $\langle TPortes_i[id_i], Portes_i[id_i]\rangle$ pour lui donner une nouvelle valeur $\neq \langle *, 0\rangle$, et chacun de ces couples est sous la forme $\langle *, 0\rangle$ (sauf s'il s'agit du contenu d'un message avec $T < TPortes_j[id_i]$ auquel cas cette valeur sera ignorée).

Si ce lemme est faux, cela signifie que pour une exécution $\mathcal{E}$ donnée, la prochaine période occupée du diviseur (après t) n'est pas *convenable* ? Donc en particulier, chaque appel à Divise retourne *bas*, ou chacun de ces appels retourne *droite*, et en particulier chaque appel termine. Considérons tout d'abord le premier cas, où tous les appels retournent *bas*.

Soit P_I l'ensemble des processus appelant *Divise* durant cette période, et soit p_{max} le processus de P_I avec le plus grand identifiant id_{max}. Pour qu'un processus p_i retourne *bas*, il doit vérifier $max(id \in Connus_i$ tels que $Portes_i[id] > 0) \neq id_i$ (ligne 19). Autrement dit, $Portes_i[id_i] = 0$ ou $\exists id_j > id_i$ tel que $Portes_i[id_j] > 0$. Le premier cas est impossible, car les valeurs de $\langle TPortes_i[id_i], Portes_i[id_i]\rangle$ sont croissantes et que la dernière a été générée par p_i lui-même ligne 14, avec $Portes_i[id_i] > 0$. Donc, si p_{max} retourne *bas*, cela signifie que $\exists id_j > id_{max}$ tel que $Portes_{max}[id_j] > 0$. D'après la première observation de ce lemme, cela signifie que p_j doit avoir appelé *Divise* après t et avant que l'appel à *Divise* par p_{max} ne retourne. Donc, p_j a invoqué *Divise* durant cette période occupée, et $id_j > id_{max}$, ce qui contredit la définition de p_{max}.

Supposons maintenant que tous les appels retournent *droite*. Pour qu'un appel à *Divise* par p_i retourne *droite*, il faut que $Portes_i[id_j] \geq r_i$ (ligne 13) pour un certain id_j. Soit r_m la plus grande ronde r telle que, pour un certain processus p_i et un certain id_j, une invocation de *Divise* par p_i pendant cette période termine en vérifiant $Portes_i[id_j] \geq r$. r_m est bien défini car pour chaque processus p_i, r_i est un entier entre 1 et N. Soit Inv une invocation de *Divise* par un processus p_i durant cette période occupée telle que lorsque *droite* est retourné (ligne 13), pour un certain id_m, $Portes_i[id_m] = r_m$. Soit p_m le processus d'identifiant id_m, ce qui signifie que p_m stocke r_m dans sa variable $Portes_m[id_m]$ ligne 14 au cours d'une invocation Inv_m à *Divise*. En effet, comme observé au début de ce lemme, $Portes_i[id_m]$ ne peut prendre une valeur $r > 0$ qu'après un appel à *Divise* par p_m au cours de la période occupée. D'après l'algorithme, p_m ne peut pas, au cours de cet appel, retourner *droite* durant les rondes $1, \ldots, r_m$. Si *droite* est retourné durant une ronde postérieure $r > r_m$, alors à cause du test ligne 13, on aurait un certain id_j vérifiant $Portes_m[id_j] = r_j \geq r > r_m$. De manière similaire, cela signifie que le processus p_j a appelé *Divise* au cours de cette période, et a stocké r_j dans sa variable $Portes_j[id_j]$. Or, au cours d'une exécution de *Divise* par p_j, la valeur de $Portes_j[id_j]$ ne fait qu'augmenter. Donc, lorsque cet appel termine, $Portes_j[id_j] \geq r_j > r_m$, ce qui contredit la définition de r_m. Donc, Inv_m ne retourne pas *droite*.

Cela signifie donc que la prochaine période occupée est nécessairement convenable. Par définition, le diviseur est donc dans une période *libre*. □

Corollaire 5.3.6. *Initialement, le diviseur est dans une période libre.*

Démonstration. En effet, initialement, chaque $Portes_i[id] = 0$, et aucun message n'est en cours. De plus, le diviseur n'est pas occupé car aucun processus n'a encore appelé Divise. Donc, d'après le lemme précédent, initialement le diviseur est *libre*, et sa première période occupée est donc *convenable*. □

Lemme 5.3.7. *Tout intervalle de temps durant lequel chaque invocation de Divise retourne bas est fini.*

Démonstration. Supposons par contradiction que pour une exécution E telle que après un certain temps t_1, *Divise* est appelé infiniment souvent, et retourne toujours *bas*. Soit p_m le processus avec le plus grand identifiant id_m parmi les processus qui invoquent *Divise* infiniment souvent. Pour chaque processus p_j avec $id_j > id_m$, p_j invoque *Divise* un nombre fini de fois. Donc, après un certain temps t_2, chaque tel p_j n'appelle plus *Divise* et soit il a terminé son dernier appel, soit il est tombé en panne. Entre le début de E et l'instant t_2, il n'y a qu'un nombre fini d'appels à *Divise* par les processus p_j, $id_j > id_m$, et donc un nombre fini de valeurs de $\langle TPortes[id_j], Portes[id_j]\rangle$ différentes.

Après t_1, chaque invocation à *Divise* par p_m retourne *bas*. Cela n'est possible que si $Portes_m[id_j] > 0$ avec $id_j > id_m$ ligne 19, pour un certain id_j. Or, avant cela, p_m vérifie que $Portes_m[id] = 0$ pour tout id (ligne 13), et dans le cas contraire retourne *droite*. Donc, la valeur de $Portes_m[id_j]$ doit changer au cours de cette appel.

On sait que les valeurs successives de $\langle TPortes_m[id], Portes_m[id]\rangle$ sont croissantes, et ce pour tout identifiant id. Comme une infinité d'invocations de *Divise* par p_m, toutes retournant *bas*, et qu'il n'y a qu'un nombre fini de processus p_j vérifiant $id_j > id_m$, et comme $Portes_m[id_j]$ doit changer pour un certain id_j à chaque appel, cela signifie qu'il existe une suite (croissante) infinie de $\langle TPortes_m[id_j], Portes_m[id_j]\rangle$ pour au moins un $id_j > id_m$. Cela contredit le fait qu'il n'y a qu'un nombre fini de $\langle TPortes[id_j], Portes[id_j]\rangle$ pour chaque $id_j > id_m$. □

Lemme 5.3.8. *À tout instant, au plus $\lfloor n/(n-f)\rfloor$ processus ont* capturé *ce diviseur, c'est-à-dire ont obtenu stop en réponse à Divise et n'ont pas encore appelé Relache.*

Démonstration. Supposons par contradiction qu'à un instant t, $s > \lfloor n/(n-f)\rfloor$ processus ont obtenu *stop* et n'ont pas encore appelé *Relache*. Soient $Inv_1, \ldots, Inv_s$ les dernières invocations à *Divise* par ces processus avant t (donc celles ayant retourné *stop*). Sans perdre en généralités, on note p_i le processus ayant exécuté Inv_i et d'identifiant id_i. Dans la ronde r de l'appel Inv_i, p_i reçoit un ensemble de $n - f$ messages RepId (ligne 17). On note $Q(r, p_i)$ l'ensemble des $n - f$ processus qui ont envoyé ces messages.

On démontre alors la propriété intermédiaire suivante : Soit $1 \leq r < r' \leq N$ et $i, i' \in [1, s]$. Si $id_{i'} < id_i$, alors $Q(r, p_i) \cap Q(r', p_{i'}) = \emptyset$.

Supposons par contradiction que $Q(r, p_i) \cap Q(r', p_{i'}) \neq \emptyset$, et soit $p_x \in Q(r, p_i) \cap Q(r', p_{i'})$.

$p_x \in Q(r, p_i)$ signifie que p_x a envoyé un message m de la forme (RepId, $\langle TP_x, P_x\rangle, *, \langle A, r\rangle$) à p_i (ligne 37). Ce message est une réponse à (Id, $\langle TP_\ell, P_\ell\rangle, *, \langle A, r\rangle$) envoyé par p_i, avec $P_\ell[id_i] = r$ (ligne 16). Donc, quand m est envoyé, $Portes_x[id_i] = P_x[id_i] = r$, car $p_x \in Q(i, r)$ et donc p_i ne peut pas avoir modifié son $\langle TPortes_i[id_i], Portes_i[id_i]\rangle$ pour une valeur plus récente.

De même, $p_x \in Q(r', p_{i'})$ signifie qu'il a envoyé un message $m' =$ (RepId, $\langle TP'_x, P'_x\rangle, *, \langle A', r'\rangle$) à $p_{i'}$ en réponse à (Id, $\langle TP'_\ell, P'_\ell\rangle, *, \langle A', r'\rangle$). Et lorsque m' est envoyé, $Portes_x[id_{i'}] = r'$.

Si m est envoyé avant m', comme la valeur de $Portes_m[id_i]$ ne décroît pas jusqu'à ce que p_i appelle *Relache*, la valeur $P'_x[id_i]$ contenue dans m' est au moins $r > 0$, avec $id_i > id_{i'}$. Donc, après que m' soit reçu par $p_{i'}$ et jusqu'à ce que p_i invoque *Relache*,

$max(id \in Connus : Portes_{i'}[id] > 0) \geq id_i > id_{i'}$, d'où on déduit que $Inv_{i'}$ retourne *bas* (ligne 19), ce qui est une contradiction.

Si m' est envoyé avant m, on a alors lorsque m est envoyé $\langle TP_x[id_{i'}], P_x[id_{i'}]\rangle \geq \langle TP'_x[id_{i'}], P'_x[id_{i'}]\rangle$. De plus, au processus p_i, lorsque m est reçu, $TPortes_i[id_{i'}] \leq TP'_x[id_{i'}]$ car $p_{i'}$ na pas encore appelé *Relache* depuis son invocation $Inv_{i'}$. Donc, après réception de m, $Portes_i[id_{i'}] \geq P'_x[id_{i'}] \geq r' > r$ (ligne 32). On peut donc en déduire que lors de la ronde $r+1$ de Inv_i, l'appel retourne *droite*, ce qui est une contradiction. Cela prouve bien cette propriété.

Supposons sans perdre en généralités que $id_1 > \ldots > id_s$. Donc, les ensembles $Q(1, p_1), \ldots, Q(N, p_N)$ sont bien définis car $s \geq N$, et sont deux à deux disjoints d'après la propriété que l'on vient de démontrer. Donc, $|\bigcup_{1\leq i\leq N} Q(i, p_i)| = N(n-f) > n$ car $N = \lfloor n/(n-f)\rfloor + 1$. Cela est impossible, car le système ne comprend que n processus. □

On a donc bien :

Théorème 5.3.9. *L'algorithme 5.7 implémente un $\lfloor n/(n-f)\rfloor$-diviseur réutilisable.*

5.3.3 k-renommage réutilisable

On s'intéresse maintenant à un algorithme permettant de résoudre le (M, k)-renommage réutilisable pour un certain M et k.

Il est possible de réaliser un tel algorithme utilisant une grille de k-diviseur réutilisables, vérifiant des propriétés supplémentaires, comme énoncé dans [30]. Cette grille est similaire à celles évoquées précédemment pour les renommages à-un-coup classique et k-redondant, ainsi que pour le renommage réutilisable classique. Cependant, la taille de cette grille doit être agrandie, et certaines propriétés supplémentaires sont souhaitées.

Premièrement, on souhaite borner, pour tout instant t, le nombre de diviseurs de la grille qui ne sont pas dans une période *libre*. Dans ce but, on suppose que l'implémentation des k-diviseurs réutilisables de cette grille vérifie une propriété caractérisant les périodes libres similaire à celle du lemme 5.3.5. De plus, on impose que les k-diviseurs réutilisables utilisés soient implémentés de manière liée, et non pas indépendamment les uns des autres. Plus précisément, on suppose sans perdre en généralités que l'on dispose d'un algorithme à informations complètes et qui fonctionne sur un modèle de communication par requête-réponse. C'est-à-dire que chaque fois qu'un message est envoyé par p_i, il contient l'état complet σ_i de p_i. Lorsqu'un tel message est reçu par p_j, ce dernier modifie son état σ_j à l'aide d'une fonction $actualiser(\sigma_j, \sigma_i)$. De plus, il existe deux sortes de messages, les Requêtes et les Réponses. Une Réponse est envoyée après réception d'un message de Requête. Lorsqu'un message de Requête est envoyé, il l'est à tous les processus à la fois (ce qu'on peut appeller *broadcast*), et l'envoyeur peut attendre de recevoir jusqu'à $n-f$ messages de Réponse correspondants. On considérera de plus qu'un tel processus attend exactement $n-f$ réponses (quitte à ce qu'il ignore les messages excédentaires), ou n'attend pas du tout. Par exemple, dans l'algorithme 5.7 d'implémentation de $\lfloor n/(n-f)\rfloor$-diviseur présenté en partie 5.3.2, les messages de type Test, Id et Relache sont des Requêtes, alors que RepTest et RepId sont des Réponses.

Un tel algorithme $\mathcal{S}$ est composé de n algorithmes locaux $\mathcal{S}_1, \ldots, \mathcal{S}_n$, un par processus. Comme on souhaite implémenter une grille de M diviseurs, chaque processus p_i exécute M occurrences de son algorithme local : $\mathcal{S}_i^1, \ldots, \mathcal{S}_i^M$. Si la grille est correctement utilisée, chaque processus p_i ne peut être occupé qu'avec au plus un diviseur ℓ à la fois. Ainsi, l'état des algorithmes locaux $\mathcal{S}_i^j$ avec $j \neq \ell$ sait que p_i n'est pas occupé avec le diviseur j, et l'état de $\mathcal{S}_i^\ell$ sait que p_i est occupé avec le diviseur ℓ. En particulier, tout état σ_i de

cet ensemble d'algorithmes locaux sait que p_i est occupé avec au plus un diviseur ℓ, et cet état est envoyé dans les messages de p_i. De plus, comme chaque état complet est mis à jour (via la fonction *actualiser*) à réception d'un message, et que les données provenant d'un processus p_j sont toutes totalement ordonnées (ses algorithmes locaux s'exécutant séquentiellement), chaque tel état σ_i *considère* (c'est-à-dire *sait*, si les données dont il dispose sont à jour) chaque processus p_j comme occupé avec au plus un diviseur. On suppose alors qu'on a le lemme suivant, généralisation du lemme 5.3.5 :

Lemme 5.3.10. *Si à un instant t, tout état σ d'un processus considère un diviseur ℓ comme non occupé, et que les états σ de chaque message* utile *(défini ci-dessous) considèrent ce diviseur ℓ comme non occupé, alors ℓ est dans une période libre.*

Un message est dit *utile* à un instant t si il est possible qu'après réception de ce message, le processus p_j destinataire modifie effectivement son état σ_j. Les Requêtes peuvent être *acceptées* (ligne 20) ou *refusées* (ligne 18), et on pourra alors dire qu'un message de Requete qui ne peut qu'être refusé n'est pas utile.

Chaque processus p_i, au lieu d'exécuter M threads indépendants en parallèle, exécute séquentiellement un pas de calcul dans chaque occurrence $\mathcal{S}_i^\ell$ l'un après l'autre (ligne 7). En particulier, à tout instant, p_i ne peut pas attendre de Réponses correspondantes à une Requête dans plus d'une occurrence (lignes 8–13).

Algorithme 5.8 Association de l'implémentation des k-diviseurs réutilisables d'une grille (code exécuté par p_i)

1: **Initialisation**
2: **pour chaque** $\ell \in [1..M]$ **faire** $\sigma_i[\ell] \leftarrow$ état initial de of $\mathcal{S}_i^\ell$;
3: $dernier\_recu_i[1..n] \leftarrow [0,..,0]$; $dernier\_envoi_i[1..n] \leftarrow [0,..,0]$;
4: $s_i \leftarrow 0$; $\ell \leftarrow 1$; $Rep_i[1,2,\ldots] \leftarrow [\emptyset,\ldots,\emptyset]$;
5: **Procédure** GRILLE
6: **tant que** *vrai* **faire**
7: $\ell \leftarrow (\ell \bmod M) + 1$; Effectuer un pas de calcul de $\mathcal{S}_i^\ell$; $\sigma_i[\ell] \leftarrow$ état de $\mathcal{S}_i^\ell$;
8: **si** le pas de $\mathcal{S}_i^\ell$ est un *broadcast* d'une Requête **alors** ▷ envoi d'une Requete à tous les processus
9: $dernier\_req_i \leftarrow \sigma_i$; $s_i \leftarrow s_i + 1$; $Rep_i[s_i] \leftarrow \emptyset$;
10: **pour chaque** $j \in [1..n]$ **faire**
11: **envoyer** (Requete, $dernier\_envoi_i[j]$, s_i, $\langle dernier\_recu_i[j], \sigma_i\rangle$) **à** p_j ;
12: **si** prochain pas de $\mathcal{S}_i^\ell$ est **Attendre** ($n - f$ réponses) **alors** ▷ Sinon, il n'attend aucune réponse
13: **attendre** que $|Rep_i[s_i]| \geq n - f$;
14: **Quand** un message $m = (type, de, s, \_)$ **est reçu** du processus p_j
15: $dernier\_recu_i[j] \leftarrow ls$;
16: **selon** la forme de m
17: **cas** $m = (\text{Requete}, de, s, \langle dr, \sigma\rangle)$ **faire**
18: **si** $dr < dernier\_envoi_i[j]$ **alors** ▷ m est refusé
19: **envoyer** (Refuse, $dernier\_envoi_i[j]$, s, $\_$) **à** p_j ;
20: **sinon** ▷ m est accepté
21: $dernier\_envoi_i[j] \leftarrow dernier\_envoi_i[j] + 1$;
22: **pour chaque** $\ell \in [1, M]$ **faire** ACTUALISER($\sigma_i[\ell], \sigma[\ell]$) ;
23: **envoyer** (Reponse, $dernier\_envoi_i[j], s, \sigma_i$) **à** p_j ;
24: **cas** $m = (\text{Reponse}, de, s, \sigma)$ **faire**
25: **si** $s = s_i$ **alors** $Rep_i[s] \leftarrow Rep_i[s] \cup \{m\}$;
26: **pour chaque** $\ell \in [1, M]$ **faire** ACTUALISER($\sigma_i[\ell], \sigma[\ell]$) ;

27: **cas** $m = (\text{Refuse}, de, s, \_)$ **faire**
28: **si** $s = s_i$ **alors**
29: **envoyer** (Requete, $dernier\_envoi_i[j]$, s_i, $\langle dernier\_req_i, dernier\_recu_i[j]\rangle$) **à** p_j ;

Lorsque, au processus p_j, une Requête est acceptée (ligne 20), une Réponse correspondante est envoyée (ligne 23) et l'état de chaque occurrence est actualisé (ligne 22). Une Requête de p_i est acceptée par p_j si (et seulement si) la dernière Réponse envoyée de p_j à p_i a été reçue par p_i avant que cette Requête ne soit envoyée. Dans ce but, un compteur $dernier\_envoi_j[i]$ identifie la Réponse envoyée par p_j à p_i. Dans le processus p_i, $dernier\_recu_i[j]$ stocke la valeur du compteur correspondant obtenu lors de la dernière Réponse reçue par p_i de p_j. Ce $dernier\_recu_i[j]$ est envoyé avec chacune des Requête de p_i à p_j, de telle sorte que p_j puisse vérifier si sa dernière Réponse a bien été reçue ou non. Ce mécanisme assure qu'à tout instant, au plus une des Requêtes envoyées par p_i et pas encore reçues par p_j peut être acceptée par p_j. De plus, comme les canaux de communication sont FIFO, et qu'un message de Réponse n'est envoyé qu'immédiatemment après qu'une Requête soit acceptée, à tout instant, au plus un message de Réponse a été envoyé par p_j et pas encore reçu par p_i. Ce système est semblable, bien que légèrement plus complexe, au système de *cles* utilisé dans l'algorithme 4.4.

Les messages de Requête envoyés par p_i qui sont refusés (ligne 18) sont jetés et ignorés par p_j. Cependant, pour empêcher que p_i attende pour toujours de recevoir $n-f$ Réponses à une de ses Requêtes, p_j envoie un message Refuse (ligne 19) demandant à ce que la Requête soit envoyée à nouveau. Lorsque le message Refuse est reçu par p_i, toute Réponse de p_j à p_i a été reçue. Donc, si p_i attend toujours ses $n-f$ Réponses, la Requête envoyée à nouveau contient la valeur portée par la dernière Réponse reçue de p_j, et elle sera donc nécessairement acceptée.

Tout processus p_a qui traverse la grille de k-diviseurs n'appelle une opération *Divise* ou *Relache* que sur un seul k-diviseur à la fois. Chaque message de Requête ou de Réponse envoyé par p_i contient non seulement l'état $\sigma_i[\ell]$ de l'occurrence $\mathcal{S}_i^\ell$ correspondant à la raison de l'envoi de ce message, mais aussi les états $\sigma_i[1], \ldots, \sigma_i[M]$ de chaque autre occurrence. Lorsqu'un tel message est reçu (et accepté, dans le cas d'une Requête) par p_j, il actualise l'état de chacune de ses occurrences $\mathcal{S}_j^1, \ldots, \mathcal{S}_j^M$ grâce aux états $\sigma_i[1], \ldots, \sigma_i[M]$ (lignes 22 et 26). En plus d'assurer la progression de chaque occurrence chez tout processus correct malgré la possibilité de Requêtes refusées, cela garantit aussi qu'au plus un des états $\sigma_i[1], \ldots, \sigma_i[M]$ est un état dans lequel le processus p_a a une opération active, pour tout processus p_a. Donc, à tout instant, d'après l'ensemble d'états σ_i, au plus n des M k-diviseurs de la grille sont considérés comme *occupés*, comme expliqué avant le lemme 5.3.10.

On sait aussi que, à tout instant, chacun des $O(n^2)$ canaux de communication contient au plus une Réponse et au plus une Requête susceptible d'être acceptée. Donc, chaque canal contient au plus 2 messages *utiles* (dans le sens où de tels messages peuvent modifier l'état des diviseurs via la fonction *actualiser*). De plus, chaque ensemble d'états σ contenu dans un tel message considère au plus n diviseurs comme étant occupés. Donc, à tout instant, au plus $O(n^3)$ k-diviseurs différents peuvent être considérés comme occupés à un moment où un autre après cet instant si aucune opération n'est appelée. En d'autres termes, chacun des autres k-diviseurs n'est pas et ne peut pas être considéré comme occupé même après réception des messages en cours, d'où :

Lemme 5.3.11. *Il existe une borne $B = O(n^3)$ telle que, à tout instant, au plus B k-diviseurs de la grille ne sont pas dans une période* libre.

Revenons au problème du (M, k)-renommage réutilisable.

On utilise le même principe de grille que pour les autres versions de renommage à base de diviseurs, ici avec une grille triangulaire dont les diviseurs ont pour identifiants $x = (a + b) * (a + b + 1)/2 + b$, comme représenté dans la figure 5.3.

Algorithme 5.9 Renommage k-redondant réutilisable à base d'une grille de k-diviseurs. (code exécuté par p_i)

```
1:  Initialisation
2:      GD est un tableau à 2 dimensions de k-diviseurs réutilisables, dont on peut accéder
        à un k-diviseur via GD[a, b], avec a, b ∈ ℕ. Ces objets sont implémentés collectivement
        comme détaillé dans l'algorithme 5.8.
3:      a_i ← 0 ; b_i ← 0 ;
4:  Fonction RENOMME(id_i)
5:      a_i ← 0 ; b_i ← 0 ;
6:      répéter
7:          d_i ← DIVISE(id_i) exécuté sur le k-diviseur GD[a_i, b_i] ;
8:          si d_i = bas alors
9:              b_i ← b_i + 1 ;
10:         sinon si d_i = droite alors
11:             a_i ← a_i + 1 ;
12:     jusqu'à ce que d_i = stop
13:     x_i ← (a_i + b_i) * (a_i + b_i + 1)/2 + b_i ;
14:     retourner x_i ;
15: Fonction LIBERE(id_i)
16:     exécute RELACHE(id_i) sur le k-diviseur GD[a_i, b_i] ;
```

Un tel algorithme garantit que les processus souhaitant obtenir un nouveau nom évoluent dans la grille, en avançant de diviseur en diviseur, toujours en incrémentant $a_i + b_i$. De plus, ils retournent une valeur x_i après avoir obtenu *stop* sur un k-diviseur de coordonnées $\langle a_i, b_i \rangle$, et cette valeur est associée de manière unique à ces coordonnées et donc à cet objet (observation 5.1.17).

On remarque que :

Lemme 5.3.12. *À tout instant, une valeur x est capturée par au plus k processus.*

Démonstration. Un processus ne capture une valeur x que lorsqu'il obtient la réponse *stop* à son appel à *Divise* sur le x-ième k-diviseur de la grille. De plus, un tel processus n'appelle *Relache* sur ce diviseur que lorsqu'il choisit de libérer sa valeur via la fonction *Libere*. Donc, si $k + 1$ processus ont capturé un même x à un même instant, cela signifie qu'ils ont tous obtenu la réponse *stop* lors de leur appel à *Divise* sur le même k-diviseur, alors qu'aucun n'a encore appelé *Relache* depuis. Cela contredit la propriété des k-diviseurs réutilisables, et est donc impossible. □

On remarque aussi que :

Lemme 5.3.13. *Tout appel à Libere par un processus correct termine.*

Démonstration. En effet, tout appel à *Relache* sur un k-diviseur par un processus correct termine. □

Il reste donc à montrer que, après avoir évolué suffisamment profondément dans la grille, tout processus finit par obtenir *stop* et donc retourner une valeur. Une borne M_0 sur $a+b$ telle qu'aucun processus ne peut accéder aux k-diviseurs de coordonnées $\langle a,b\rangle$ avec $a+b \geq M_0$ suffit à le démontrer, et à donner une borne $M = M_0 * (M_0+1)/2$ sur la taille de l'espace des noms.

On considère la grille GD comme un ensemble de diagonales $D_0, D_1, \ldots$, tels que la diagonale D_d contient exactement les diviseurs de coordonnées $\langle a,b\rangle$ avec $a+b=d$. On pourra remarquer que, lorsqu'un processus évolue dans la grille, il passe à chaque changement de diviseur d'une diagonale D_d à la suivante D_{d+1}. On dira qu'un processus p_i est *contenu* dans une diagonale D_d à un instant t si p_i est occupé avec un diviseur de D_d.Pour $d \leq d'$, on notera $[D_d, D_{d'}]$ l'ensemble des diagonales comprises entre D_d et $D_{d'}$, c'est-à-dire l'ensemble des diviseurs $\langle a,b\rangle$ avec $d \leq a+b \leq d'$. On dira alors qu'un processus p_i est contenu dans $[D_d, D_{d'}]$ à un instant t s'il est contenu dans une diagonale D_e avec $d \leq e \leq d'$. Par extension, $[D_d, \infty]$ désignera l'ensemble des diagonales D_e de la grille avec $e \geq d$.

On rappelle que B désigne une borne sur le nombre maximal de k-diviseurs qui, à un instant t donné, ne sont pas dans une période libre.

Lemme 5.3.14. *Soit $p \in [1,n]$, et $d_1 < d_2$ tel que $d_2 - d_1 \geq B + p$. Supposons qu'à tout instant, au plus p processus sont contenus dans $[D_{d_1}, \infty]$. Alors, à tout instant, au plus $p-1$ processus sont contenus dans $[D_{d_2}, \infty]$.*

Démonstration. Supposons par contradiction qu'il existe une exécution et un instant t_f durant lequel les p processus contenus dans $[D_{d_1}, \infty]$ sont contenus dans $[D_{d_2}, \infty]$ (comme $d_2 > d_1$, $[D_{d_2}, \infty] \subset [D_{d_1}, \infty]$). On peut noter que, pour être contenu dans une diagonale D_d avec $d \geq d_2$, un processus p_i doit appeler *Divise* sur des diviseurs des diagonales $D_0, D_1, \ldots, D_{d-1}$ dans cet ordre (et obtenir *droite* ou *bas* à chaque fois), et donc en particulier il doit à un moment être contenu dans D_{d_1}. Soit t_0 le dernier instant avant t_f durant lequel un processus est contenu dans la diagonale D_{d_1} (à t_f, aucun processus ne peut y être contenu).

Soit $O \subseteq [D_{d_1}, D_{d_2}]$ l'ensemble des k-diviseurs qui sont occupés à un moment entre t_0 et t_f. Soit n_d le nombre de diviseurs de la diagonale D_d qui sont aussi dans O, c'est-à-dire $n_d = |O \cap D_d|$. On veut maintenant démontrer que, pour toute diagonale D_d avec $d_1 \leq d < d_2$, on a $n_{d+1} - n_d \geq 1 - NL_d$, où NL_d est le nombre de diviseurs de la diagonale D_d qui ne sont pas en période libre à l'instant t_0.

Si un diviseur $\delta \in O$ et dans la diagonale D_d ($d_1 \leq d < d_2$) est libre à t_0, alors il ne peut pas passer dans une période non-libre à moins qu'un processus appelle *Divise* sur δ. Cela signifie en particulier qu'un processus qui appelle *Divise* de cette manière obtient *stop*, ou qu'au moins 2 processus appellent *Divise* sur δ dans la période $[t_0, t_f]$. Comme les p processus considérés atteignent tous au moins la diagonale D_{d_2}, aucun processus n'obtient *stop* d'un tel diviseur $\delta \in [D_{d_1}, D_{d_2-1}]$ durant $[t_0, t_f]$, car dans ce cas ce processus ou un autre devrait alors entrer dans $[D_{d_1}, \infty]$ avant t_f (pour atteindre à nouveau les p processus), ce qui contredit la définition de t_0. Donc, si δ était libre à t_0 et qu'il devient non-libre avant t_f, alors au moins 2 processus appellent *Divise* sur δ durant $[t_0, t_f]$.

De plus, comme δ était libre avant le premier appel à *Divise* durant $[t_0, t_f]$ et que *stop* ne peut pas être retourné, alors *droite* et *bas* sont tous deux retournés par δ à un moment durant $[t_0, t_f]$. En conséquence, les deux diviseurs qui suivent δ dans la diagonale suivante D_{d+1} (c'est-à-dire les diviseurs $\langle a+1, b\rangle$ et $\langle a, b+1\rangle$ si δ a pour coordonnées $\langle a,b\rangle$) vont être occupés, et appartiennent donc à O. En effet, s'ils ne sont jamais occupés avant t_f, cela signifie que le processus ayant obtenu *droite* ou *bas* de δ n'a jamais appelé *Divise* après ce retour, ou que l'appel sur δ n'a pas encore retourné, ce qui contredit le fait que ce processus doit être contenu dans $[D_{d_2}, \infty]$ à t_f.

Si un diviseur $\delta' \in O$ est non-libre à t_0, alors il est possible que toute invocation de *Divise* sur δ' retourne une même direction *droite* ou *bas* durant $[t_0, t_f]$, mais aucun de ces appels ne peut retourner *stop* (comme évoqué précédemment, par définition de t_0). Donc, si $\delta' \in D_d$ avec $d_1 \leq d < d_2$, alors au moins un des deux diviseurs suivants δ' est occupé durant $[t_0, t_f]$ (c'est-à-dire le diviseur $\langle a+1, b\rangle$ ou $\langle a, b+1\rangle$, en notant $\langle a, b\rangle$ les coordonnées de δ'). D'où au moins un de ces deux diviseurs appartient à O.

Ensuite, par induction sur NL_d, on démontre que $n_{d+1} - n_d \geq 1 - NL_d$. En effet, pour $NL_d = 0$, chaque diviseur $\langle a, b\rangle \in O \cap D_d$ vérifie que ses deux successeurs $\langle a+1, b\rangle$ et $\langle a, b+1\rangle$ sont dans O, ce qui signifie qu'au moins n_d diviseurs différents de la forme $\langle a+1, b\rangle$ (avec $a+b=d$) sont dans O. De plus, le diviseur $\langle a_0, b_0+1\rangle$, avec le a_0 minimal parmi les a des diviseurs $\langle a, b\rangle \in O \cap D_d$, est aussi dans O et ne fait pas partie des n_d de la forme $\langle a+1, b\rangle$. Donc, pour $NL_d = 0$, $n_{d+1} - n_d \geq 1$.

De plus, si cette propriété est vraie pour $NL_d = X$, alors elle est aussi vraie pour $NL_d = X+1$. En effet, la seule différence est qu'au plus un des diviseurs de $O \cap D_d$ est dans une période non-libre (au lieu de libre) à t_0. Donc, pour un tel diviseur, il est possible qu'un seul de ses deux successeurs soit dans O au lieu des deux, ce qui diminue le nombre de diviseurs dans $O \cap D_{d+1}$ d'au plus 1 par rapport au cas où $NL_d = X$. Donc, si dans le premier cas on avait $n^X_{d+1} - n_d \geq 1 - X$, on a maintenant $n^{X+1}_{d+1} \geq n^X_{d+1} - 1$ d'où $n^{X+1}_{d+1} - n_d \geq 1 - X - 1 = 1 - (X+1)$, ce qui démontre bien cette propriété.

En additionnant ces inégalités $n_{d+1} - n_d \geq 1 - NL_d$, on obtient $n_{d_2} - n_{d_1} \geq d_2 - d_1 - NL_{Tous}$, où NL_{Tous} est le nombre de diviseurs de $[D_{d_1}, D_{d_2}]$ qui sont non-libres à t_0. Comme (par définition de B), $NL_{Tous} \leq B$, on a $n_{d_2} \geq n_{d_1} + d_2 - d_1 - B \geq n_{d_1} + B + p - B$ car $d_2 - d_1 \geq B + p$. Donc, $n_{d_2} \geq p+1$, car au moins un diviseur de la diagonale D_{d_1} est occupé à t_0 (par définition de t_0), d'où $n_{d_1} \geq 1$.

Mais, lors de cet intervalle $[t_0, t_f]$, pas plus de p processus ne peuvent être contenus dans $[D_{d_1}, \infty]$, et chacun de ces processus évolue d'une diagonale à la suivante. Comme un processus ne peut ainsi pas entrer (c'est-à-dire appeler *Divise*) dans plus d'un diviseur par diagonale, au plus p diviseurs de la diagonale D_{d_2} sont dans O. Ce qui signifie que $n_{d_2} \leq p$, qui est une contradiction. □

Lemme 5.3.15. *Soit $p \in \mathbb{N}$ et $C_l = B + l$ pour tout $l \geq 1$. Soit $C^p = \sum_{l=1}^{p}(C_l) = O(B * p)$. Si au plus p processus utilisent la grille de k-diviseurs GD, alors aucun processus ne peut atteindre la diagonale D_{C^p}.*

Démonstration. D'après le lemme précédent, on sait que, lors de toute exécution et à tout instant, au plus $p-1$ processus peuvent être contenus dans $[D_{C_p}, \infty]$. En appliquant à nouveau ce lemme, on sait qu'au plus $p-2$ processus peuvent être contenus dans $[D_{C_p + C_{p-1}}, \infty]$, car $C_p + C_{p-1} - C_p = B + p - 1$ et qu'au plus $p-1$ processus sont contenus dans $[D_{C_p}, \infty]$. Par récurrence, on montre qu'au plus $X-1$ processus sont contenus dans $[D_{\sum_{l=X}^{p} C_l}, \infty]$. On a donc au plus 0 processus dans $[D_{\sum_{l=1}^{p} C_l}, \infty]$, ce qui signifie bien qu'aucun processus ne peut atteindre les diviseurs de la diagonale D_{C^p}. □

Donc, cela signifie que tout processus correct qui exécute l'algorithme de renommage obtient *stop* dans un diviseur $\langle a, b\rangle$ avec $a + b < C^n$, c'est-à-dire que l'espace des noms ainsi formé est $[0, C^n(C^n+1)/2 - 1]$. Or, $C^n = O(B * n) = O(n^4)$, d'où :

Théorème 5.3.16. *Cet algorithme est un algorithme de (M, k)-renommage réutilisable avec $M = C^n(C^n+1)/2 = O(n^8)$.*

En particulier, comme un algorithme implémentant des $\lfloor n/(n-f)\rfloor$-diviseurs réutilisables a été présenté dans la partie 5.3.2, on peut faire que cet algorithme de (M, k)-renommage le soit avec $k = \lfloor n/(n-f)\rfloor$.

Chapitre 6

Conclusion

6.1 Bilan

Comme expliqué dans le chapitre 1, le modèle asynchrone par envoi de messages et avec pannes est très connu et utilisé en algorithmique distribuée. Cependant, la plupart des résultats ne s'appliquent qu'en présence d'une stricte minorité de pannes potentielles. C'est pourquoi l'étude de problème classiques dans ce modèle avec une majorité de pannes est intéressante, et constitue la problématique de cette thèse.

Avant de chercher à étendre et résoudre des problèmes dans ce modèle à majorité de pannes, il convient de bien définir les différentes notions utilisées et le modèle considéré, ce qui est fait dans le chapitre 2.

Le chapitre 3 permet alors de décrire certains des problèmes classiques considérés, ainsi que les différentes approches existantes pour appréhender ce cas de la majorité de pannes. Les approches habituelles sont basées sur des hypothèses supplémentaires ou des études *pratiques* ne vérifiant pas de bonnes garanties *théoriques*, et ne sont donc pas satisfaisantes du point de vue de cette thèse, qui cherche à définir les propriétés qu'il est possible de garantir tout en se restreignant à ce modèle classique. Enfin, le chapitre 3 décrit notamment le résultat qu'est la simulation ABD de registres [16], car les résultats du chapitre 4, ainsi que certains aspects du chapitre 5, s'inspirent fortement de cette simulation.

Dans le chapitre 4, on s'intéresse à l'extension des registres atomiques en nouveaux objets similaires, qui peuvent être simulés dans le modèle par envoi de messages avec une majorité de pannes. En effet, les registres atomiques classiques peuvent être implémentés dans ce modèle si seule une stricte minorité de processus peuvent tomber en panne : c'est la simulation ABD [16].

Dans un premier temps, on considère des *bancs de registres χ-colorés*, qui sont des ensembles de χ objets semblables à des registres, si ce n'est qu'ils ne vérifient pas la propriété de *terminaison* (ou *disponibilité*). Cependant, au moins 1 de ces χ registres vérifie cette propriété, ce qui permet à cet objet d'être utile en utilisant en parallèle les différents registres du banc. On montre alors que, si les processus partagent une même fonction de χ-coloration de quorum, alors ils peuvent implémenter un banc de registres χ-colorés, malgré une majorité de pannes. De plus, si les processus connaissent initialement les identifiants de chaque processus du système, alors ils peuvent implémenter une fonction de χ-coloration de quorum avec $\chi = 2f - n + 2$. Enfin, une telle fonction de χ-coloration de quorum ne peut pas exister avec $\chi < 2f - n + 2$. Finalement, la partie 4.1.4 décrit comment correctement utiliser de tels objets, afin d'éviter les incohérences possibles liées à une utilisation naïve, et donc les contraintes à respecter pour vérifier de bonnes propriétés.

Dans un second temps, on considère des α-registres, qui sont des objets semblables

aux registres habituels, mais qui ne vérifie pas les propriétés de *cohérence* classiques. Plus précisément, en plus de propriétés de cohérence locales, et d'une propriété de *propagation* empêchant la trivialité de l'objet, il vérifie la propriété de *lecture α-bornée*, qui limite les lectures de valeurs obsolètes à au plus α différentes valeurs (voir page 38 pour plus de précisions). Malheureusement, il n'est pas possible de borner l'*âge* des valeurs lues (c'est-à-dire le nombre d'opérations d'écriture qui ont eu lieu depuis celle ayant produit cette valeur), et l'on ne peut donc guère améliorer cette propriété, qui est néanmoins non-triviale, et n'est pas vérifiée (car $\alpha = \infty$) pour la simulation ABD qui serait utilisée en présence d'une majorité de pannes. On présente alors deux algorithmes implémentant des α-registres avec $\alpha = 2(2f-n+2)-1$. Ensuite, une borne inférieure est démontrée, prouvant ainsi qu'aucun algorithme ne peut implémenter un α-registre avec $\alpha < (2f - n + 2) + 1$. Enfin, l'impossibilité de tels registres multi-écrivains est évoquée, ainsi que la contrepartie de pouvoir utiliser des vecteurs d'α-registres mono-écrivain.

En conclusion de ce chapitre, on peut remarquer que, de manière surprenante, la formule $2f-n+2$ apparaît à la fois dans l'implémentation des bancs de registres χ-colorés et dans l'implémentation et la borne inférieure des α-registres, alors que les approches et méthodes utilisées sont bien différentes.

Dans le chapitre 5, on s'intéresse à l'extension du problème du *renommage* et des objets partagés que sont les *diviseurs*. Le problème classique du renommage [17] ne peut être résolu non-trivialement dans ce modèle qu'en présence d'une stricte minorité de pannes, et ce pour sa version à-un-coup comme pour sa version réutilisable. Les diviseurs [78], utiles pour résoudre simplement le renommage, peuvent aussi être implémentés dans le modèle asynchrone par envoi de messages, mais seulement avec une stricte minorité de pannes. Ce chapitre présente donc d'abord des versions habituelles d'algorithmes de renommage et d'implémentation de diviseurs, car les résultats étendus s'en inspire, et que cela favorise la compréhension des parties suivantes.

Dans un premier temps, on s'intéresse alors aux extensions des versions *à-un-coup* de ces problèmes et objets. On définit alors intuitivement le *renommage k-redondant*, pour lequel chaque nom peut être choisi par au plus k processus au lieu d'au plus 1 pour le renommage classique. De manière similaire, les *k-diviseurs* sont définis, vérifiant qu'au plus k processus obtiennent *stop* au lieu d'au plus 1. On démontre alors qu'il est impossible de résoudre le k-renommage et d'implémenter des k-diviseurs avec $k < \lfloor n/(n-f) \rfloor$. Puis on donne des algorithmes permettant de résoudre le k-renommage et d'implémenter des k-diviseurs avec $k = \lfloor n/(n-f) \rfloor$, qui est donc optimal. Ces k-diviseurs sont d'autant plus intéressants que tout algorithme de renommage classique basé sur des diviseurs (par exemple [15]) est un algorithme de k-renommage si on remplace les diviseurs par des k-diviseurs. Enfin, on montre aussi que l'algorithme de renommage de [17] est aussi un algorithme de k-renommage avec $k = \lfloor n/(n-f) \rfloor$ si utilisé en présence d'une majorité de pannes, et que son espace de noms a une taille de $n + f$ seulement.

Dans un second temps, on considère les extensions des versions *réutilisables* de ces problèmes et objets. Le renommage réutilisable k-redondant est défini comme précédemment, en permettant à chaque nom d'être capturé par au plus k processus à tout instant, au lieu d'au plus 1. Cependant, les k-diviseurs réutilisables ne sont pas qu'une simple et intuitive extension des diviseurs réutilisables classiques. En effet, même s'il vérifient effectivement qu'au plus k processus peuvent capturer le diviseur au lieu d'au plus 1, toutes les périodes occupées ne vérifient pas nécessairement la propriété classique garantissant que tous les appels ne retournent pas la même direction *bas* ou *droite*. Grâce à cet affaiblissement d'une propriété définissant cet extension, il est possible d'implémenter ces objets avec $k = \lfloor n/(n-f) \rfloor$, ce qui est (une fois de plus) optimal. Cependant, les résultats habituels de renommage réutilisable basés sur des diviseurs réutilisables classiques [78] ne

s'appliquent plus avec les k-diviseurs réutilisables, à cause de cette propriété affaiblie. On démontre néanmoins qu'il est possible de résoudre un tel problème à l'aide de tels objets, permettant ainsi de résoudre le k-renommage avec un $k = \lfloor n/(n-f) \rfloor$ optimal. Cependant, ce résultat produit un algorithme de k-renommage avec un espace de noms de taille $O(n^8)$, ce qui est moins satisfaisant que la taille $O(n^2)$ de l'espace de noms de l'algorithme de renommage réutilisable semblable.

Enfin, bien que ces extensions réutilisables soient moins satisfaisants que ceux à-un-coup, il est intéressant de noter que dans tous les cas, on obtient un $k = \lfloor n/(n-f) \rfloor$, qui est de plus une borne optimale, car il est impossible de faire moins, ce qui justifie d'ailleurs la borne de la majorité de pannes, car $k = 1 \Leftrightarrow f < n/2$.

Finalement, un résultat intéressant apparaît de ces travaux : la différence entre la formule $2f - n + 2$ d'un côté et $\lfloor n/(n-f) \rfloor$ de l'autre. Lorsque $n/2 < f < n-1$, ces deux formules sont distinctes, et vérifient $\lfloor n/(n-f) \rfloor < 2f - n + 2$, et elles sont égales pour les cas *extrêmes* $f = n/2$ (barrière de la majorité de pannes) et $f = n-1$ (nombre maximal de pannes). L'apparition de ces deux bornes différentes souligne bien que l'extension de problèmes et objets classiques qui n'existent qu'en présence d'une stricte minorité de pannes est non-triviale, et digne d'intérêt. De tels résultats ne peuvent donc pas nécessairement être appliqués de manière similaire à d'autres problèmes, et de telles études indépendantes de problèmes individuels sont donc utiles et appréciables.

6.2 Perspectives et ouvertures

Les bancs de registres χ-colorés présentés dans le chapitre 4 sont mono-écrivains et multi-lecteurs. Une propriété (disponibilité partielle coordonnée) est cependant ajoutée afin d'utiliser plusieurs de ces bancs dans un même algorithme, chaque banc pouvant avoir un écrivain associé différent. Cependant, dans certains algorithmes, il est plus pratique d'utiliser des registres multi-écrivains. Il peut donc être intéressant d'étudier l'implémentation de bancs de registres χ-colorés multi-écrivains, qui pourraient ainsi être utilisés dans l'extension de tels algorithmes dans le modèle avec une majorité de pannes. Il est probable qu'une telle implémentation soit possible, basée sur le même principe de fonction de χ-coloration de quorums, permettant de garantir l'intersection de quorums d'une même couleur.

Les α-registres, présentés aussi dans le chapitre 4, ne peuvent pas être multi-écrivains. Il reste cependant possible d'utiliser des vecteurs de plusieurs α-registres mono-écrivains, de sorte que chaque processus ait un α-registre pour lequel il est écrivain. Si le vecteur utilise des α-registres implémentés indépendamment, chaque α-registre vérifie ses propriétés habituelles, mais aucune propriété supplémentaire n'apparaît pour le vecteur dans son ensemble. Cependant, si les α-registres sont implémentés conjointement, à la façon dont l'algorithme 5.8 lie l'implémentation des k-diviseurs réutilisables de la grille, il est probable que le vecteur vérifie des propriétés supplémentaires. Il serait donc intéressant d'étudier un tel objet dans ces conditions, par exemple pour déterminer le nombre de *combinaisons* de vieilles valeurs qu'il est possible de lire, et qui est probablement inférieur au α^n basique. Enfin, il serait intéressant de se pencher sur une extension de l'abstraction du *snapshot* sur un tel vecteur d'α-registres.

Les résultats de k-renommage et k-diviseurs présentés dans le chapitre 5 sont plutôt satisfaisants. Néanmoins, bien qu'une borne inférieure sur la valeur de k soit fournie (et atteinte, avec $k = \lfloor n/(n-f) \rfloor$), aucune borne inférieure sur la taille de l'espace des noms n'est présentée. Il pourrait être intéressant d'étudier une telle borne, notamment pour savoir si la borne inférieure de $n + f$ [17] pour le renommage à-un-coup classique

s'étend au k-renommage à-un-coup. De plus, le k-renommage réutilisable semble être plus complexe que sa version à-un-coup, et il se pourrait que la borne inférieure de son espace de noms soit plus élevée que pour le renommage réutilisable classique.

D'autre part, l'algorithme de k-renommage réutilisable fourni dans cette thèse a un espace de noms (démontré) de taille $O(n^8)$, ce qui semble exagérément élevé. Un travail cherchant à obtenir un meilleur algorithme, ou, alternativement, à améliorer cette valeur en modifiant sa démonstration, serait donc désirable.

Enfin, les k-diviseurs réutilisables ont une définition moins satisfaisante que les autres extensions du chapitre 5, car une des propriétés du diviseur réutilisable classique est nettement affaiblie. Cependant, il n'est pas possible d'implémenter des k-diviseurs réutilisables pour lesquels cette propriété serait conservée identique à la version habituelle. Malgré tout, il pourrait être profitable de rechercher des propriétés supplémentaires qu'il est possible de garantir, afin de renforcer la définition des k-diviseurs réutilisables, et de les rendre plus facilement utilisables.

Ces nombreux résultats variés montrent bien que cette étude de problèmes dans le cas d'une majorité de pannes est désirable. Ainsi, chercher à étendre et analyser d'autres problèmes et objets qui sont connus pour être résolus et implémentés avec une stricte minorité de pannes, mais qui sont impossibles avec une majorité de pannes, serait très intéressant. Il est d'ailleurs possible qu'une telle étude de problèmes indépendants amène à une meilleure compréhension de ces problèmes, et peut-être à une généralisation des propriétés des problèmes au delà de la barrière de la majorité de pannes.

Un exemple de problème intéressant à étudier est celui de l'accord approximatif, évoqué dans la partie 3.3.3. En effet, ce problème peut être résolu en présence d'une stricte minorité de pannes, mais pas en présence d'une majorité de pannes potentielles. Il est cependant simple d'étendre la définition d'accord approximatif en k-accord approximatif. De cette manière, au lieu d'imposer que toutes les valeurs soient proches à ε près (c'est-à-dire toutes contenues dans une boule de diamètre ε), on demande à ce que les valeurs puissent être réparties en k ensembles tels que les valeurs d'un ensemble sont proches entre elles à ε près (c'est-à-dire que k boules de diamètre ε chacune permettent de couvrir l'ensemble des valeurs de retour). Considérons une version non-triviale de ce problème étendu, c'est-à-dire que les valeurs initiales des processus peuvent comprendre au moins $k+1$ valeurs deux à deux distantes de plus de ε, car sinon chaque processus peut conserver sa valeur initiale et vérifier la propriété souhaitée. Il est aisé de démontrer, de manière similaire au renommage k-redondant et aux k-diviseurs (lemmes 5.2.1 et 5.2.2), qu'il est impossible de résoudre cette version non-triviale du k-accord approximatif avec $k < \lfloor n/(n-f) \rfloor$. De plus, il est trivial de le résoudre pour $k = n$. On peut alors se demander y a-t-il une borne inférieure sur ce k, plus grande que $\lfloor n/(n-f) \rfloor$, qui empêche néanmoins ce problème d'être résolu ? Et est-il possible de résoudre ce problème avec un k inférieur à n, et sous quelles conditions supplémentaires si nécessaire ? Répondre à de telles questions permettrait de produire des résultats intéressants qui resteraient bien dans la continuité des travaux de cette thèse.

Bibliographie

[1] Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *Proceedings of the 8th international conference on Principles of Distributed Systems*, OPODIS'04, pages 229–239. Springer-Verlag, 2005.

[2] Y. Afek, E. Gafni, S. Rajsbaum, M. Raynal, and C. Travers. The k-simultaneous consensus problem.

[3] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4) :873–890, September 1993.

[4] Yehuda Afek, Hagit Attiya, Arie Fouren, Gideon Stupp, and Dan Touitou. Long-lived renaming made adaptive. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 91–103. ACM, 1999.

[5] Yehuda Afek and Eli Gafni. Asynchrony from synchrony. In Davide Frey, Michel Raynal, Saswati Sarkar, RudrapatnaK. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking*, volume 7730 of *Lecture Notes in Computer Science*, pages 225–239. Springer Berlin Heidelberg, 2013.

[6] Yehuda Afek, Iftah Gamzu, Irit Levy, Michael Merritt, and Gadi Taubenfeld. Group renaming. In *Principles of Distributed Systems*, pages 58–72. Springer, 2008.

[7] Yehuda Afek and Michael Merritt. Fast, wait-free (2 k-1)-renaming. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 105–112. ACM, 1999.

[8] Yehuda Afek, Gideon Stupp, and Dan Touitou. Long lived adaptive splitter and applications. *Distrib. Comput.*, 15(2) :67–86, April 2002.

[9] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory : Definitions, implementation, and programming. *Distributed Computing*, 9(1) :37–49, 1995.

[10] Amitanand Aiyer, Lorenzo Alvisi, and RidaA. Bazzi. On the availability of non-strict quorum systems. In Pierre Fraigniaud, editor, *Distributed Computing*, volume 3724 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2005.

[11] AmitanandS. Aiyer, Lorenzo Alvisi, and RidaA. Bazzi. Byzantine and multi-writer k-quorums. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 443–458. Springer Berlin Heidelberg, 2006.

[12] Sérgio Almeida, Joao Leitao, and Luís Rodrigues. Chainreaction : a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[13] James H Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Distributed Computing*, pages 29–43. Springer, 2000.

[14] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 82–93, New York, NY, USA, 1980. ACM.

[15] James Aspnes. Slightly smaller splitter networks. *CoRR*, abs/1011.3170, 2010.

[16] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1) :124–142, January 1995.

[17] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3) :524–548, July 1990.

[18] Hagit Attiya and Arie Fouren. Polynomial and adaptive long-lived (2k-1)-renaming. In *Distributed Computing*, pages 149–163. Springer, 2000.

[19] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2) :642–664, 2001.

[20] Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks (extended abstract). In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 322–343, London, UK, UK, 1996. Springer-Verlag.

[21] Hagit Attiya and Jennifer Welch. *Distributed Computing : Fundamentals, Simulations, and Advanced Topics*. Wiley, 2 edition, 2004.

[22] Peter Bailis and Ali Ghodsi. Eventual consistency today : limitations, extensions, and beyond. *Communications of the ACM*, 56(5) :55–63, 2013.

[23] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[24] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8) :776–787, April 2012.

[25] Amotz Bar-Noy, Danny Dolev, Daphne Koller, and David Peleg. Fault-tolerant critical section management in asynchronous environments. *Information and Computation*, 95(1) :1–20, 1991.

[26] Daniel Barbara and Hector Garcia-Molina. The vulnerability of vote assignments. *ACM Trans. Comput. Syst.*, 4(3) :187–213, August 1986.

[27] David Bermbach and Stefan Tai. Eventual consistency : How soon is eventual ? an evaluation of amazon s3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, page 1. ACM, 2011.

[28] Philip A. Bernstein and Sudipto Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 923–928, New York, NY, USA, 2013. ACM.

[29] David Bonnin and Corentin Travers. α-register. In Roberto Baldoni, Nicolas Nisse, and Maarten Steen, editors, *Principles of Distributed Systems*, volume 8304 of *LNCS*, pages 53–67. Springer, 2013.

[30] David Bonnin and Corentin Travers. Splitting and renaming with a majority of faulty processes. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, pages 26 :1–26 :10, New York, NY, USA, 2015. ACM.

[31] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM.

[32] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 41–51, New York, NY, USA, 1993. ACM.

[33] Z. Bouzid and C. Travers. Parallel consensus is harder than set agreement in message passing. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 611–620, July 2013.

[34] Zohir Bouzid and Corentin Travers. (anti$-\Omega^x \times \Sigma_z$)-based k-set agreement algorithms. *Proceedings of the 14th international conference on Principles of distributed systems*, pages 189–204, 2010.

[35] Christian Cachin, Abhi Shelat, Alexander Shraer, and Er Shraer. Efficient fork-linearizable access to untrusted shared memory. In *TR RZ3688, IBM Research*, 2007.

[36] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, pages 295–304. ACM, 2008.

[37] Armando Castaneda, Sergio Rajsbaum, and Michel Raynal. The renaming problem in shared memory systems : an introduction. *Computer Science Review*, 5(3) :229–251, August 2011.

[38] Jérémie Chalopin, Emmanuel Godard, and Antoine Naudin. Anonymous graph exploration with binoculars. *arXiv preprint arXiv :1505.00599*, 2015.

[39] Sagar Chordia, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam, and Kapil Vaswani. Asynchronous resilient linearizability. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 164–178. Springer Berlin Heidelberg, 2013.

[40] Mark de Longueville. 25 years proof of the kneser conjecture the advent of topological combinatorics. *EMS Newsletter*, 53 :16–19, 2004.

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6) :205–220, October 2007.

[42] Carole Delporte-Gallet, Hugues Fauconnier, and Rachid Guerraoui. Shared memory vs message passing. Technical report, 2003.

[43] Danny Dolev, Nancy A Lynch, Shlomit S Pinter, Eugene W Stark, and William E Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM (JACM)*, 33(3) :499–516, 1986.

[44] A. D. Fekete. Asynchronous approximate agreement. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 64–76. ACM, 1987.

[45] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theor. Comput. Sci.*, 220(1) :113–156, June 1999.

[46] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, April 1985.

[47] Davide Frey, Achour Mostefaoui, Matthieu Perrin, and François Taïani. D.1.1 – Survey on Weak Consistency Approaches for Large-Scale Systems. Technical Report D1.1, LINA-University of Nantes ; IRISA, June 2015.

[48] Roy Friedman, Michel Raynal, and Corentin Travers. Two abstractions for implementing atomic objects in dynamic systems. In J.H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 73–87. Springer Berlin Heidelberg, 2006.

[49] Eli Gafni. Round-by-round fault detectors : unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, PODC '98, pages 143–152. ACM, 1998.

[50] Eli Gafni and Rachid Guerraoui. Generalized universality. In *Proceedings of the 22nd international conference on Concurrency theory*, CONCUR'11, pages 17–27. Springer-Verlag, 2011.

[51] Eli Gafni, Achour Mostéfaoui, Michel Raynal, and Corentin Travers. From adaptive renaming to set agreement. *Theoretical Computer Science*, 410(14) :1328–1335, 2009.

[52] Eli Gafni, Sergio Rajsbaum, and Maurice Herlihy. Subconsensus tasks : renaming is weaker than set agreement. In *Distributed Computing*, pages 329–338. Springer, 2006.

[53] Eli Gafni, Michel Raynal, and Corentin Travers. Test & set, adaptive renaming and set agreement : a guided visit to asynchronous computability. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 93–102. IEEE, 2007.

[54] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2) :51–59, June 2002.

[55] Seth Gilbert and Nancy Ann Lynch. Perspectives on the cap theorem. Institute of Electrical and Electronics Engineers, 2012.

[56] Joshua E Greene. A new short proof of kneser's conjecture. *American Mathematical Monthly*, pages 918–920, 2002.

[57] Pat Helland and David Campbell. Building on quicksand. *arXiv preprint arXiv :0909.1788*, 2009.

[58] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1) :124–149, January 1991.

[59] Maurice Herlihy, Sergio Rajsbaum, and Michel Raynal. Computability in distributed computing : a tutorial. *ACM SIGACT News*, 43(3) :80–102, September 2012.

[60] Maurice P Herlihy and Jeannette M Wing. Linearizability : A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3) :463–492, 1990.

[61] Damien Imbs and Michel Raynal. On adaptive renaming under eventually limited contention. In *Stabilization, Safety, and Security of Distributed Systems*, pages 377–387. Springer, 2010.

[62] Tommy R Jensen and Bjarne Toft. *Graph coloring problems*, volume 39. John Wiley & Sons, 2011.

[63] Kinga Kiss Iakab, Christian Storm, and Oliver Theel. Consistency-driven probabilistic quorum system construction for improving operation availability. In Krishna Kant, SriramV. Pemmaraju, KrishnaM. Sivalingam, and Jie Wu, editors, *Distributed Computing and Networking*, volume 5935 of *Lecture Notes in Computer Science*, pages 446–458. Springer Berlin Heidelberg, 2010.

[64] Martin Kneser. Aufgabe 360. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 2 :27, 1955.

[65] Avinash Lakshman and Prashant Malik. Cassandra : a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2) :35–40, 2010.

[66] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, July 1978.

[67] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9) :690–691, 1979.

[68] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2) :77–85, 1986.

[69] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1) :1–11, January 1987.

[70] DannyB. Lange. Mobile objects and mobile agents : The future of distributed computing ? In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1998.

[71] Eugene L Lawler and J Michael Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16(1) :77–84, 1969.

[72] Richard J Lipton and Jonathan S Sandberg. *PRAM : A scalable shared memory.* Princeton University, Department of Computer Science, 1988.

[73] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual : scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[74] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4) :203–213, 1998.

[75] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems, 2001.

[76] Hammurabi Mendes and Maurice Herlihy. Multidimensional approximate agreement in byzantine asynchronous systems. In *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*, STOC '13, pages 391–400. ACM, 2013.

[77] Hammurabi Mendes, Christine Tasson, and Maurice Herlihy. The topology of asynchronous byzantine colorless tasks. *CoRR*, 2013.

[78] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1) :1 – 39, 1995.

[79] Mark Moir and JuanA. Garay. Fast, long-lived renaming improved and simplified. In Özalp Babaoğlu and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer Berlin Heidelberg, 1996.

[80] Achour Mostefaoui, Michel Raynal, and Frédéric Tronel. From binary consensus to multivalued consensus in asynchronous message-passing systems. *Information Processing Letters*, 73(5) :207–212, 2000.

[81] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2) :423–447, April 1998.

[82] Michael Okun, Amnon Barak, and Eli Gafni. Renaming in synchronous message passing systems with byzantine failures. *Distributed Computing*, 20(6) :403–413, 2008.

[83] Vincent D Park and M Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 3, pages 1405–1413. IEEE, 1997.

[84] Jr. Parker, D.S., Gerald J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, SE-9(3) :240–247, May 1983.

[85] M. Raynal and D. Beeson. *Algorithms for Mutual Exclusion.* MIT Press, Cambridge, MA, USA, 1986.

[86] Aleta Ricciardi, Andrd Schiper, and Kenneth Birman. Understanding partitions and theno partition'assumption. In *Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of*, pages 354–360. IEEE, 1993.

[87] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, January 2011.

[88] Santosh Kumar Shrivastava and Jean-Pierre Banatre. Reliable resource allocation betvveen unreliable processes. *Software Engineering, IEEE Transactions on*, (3) :230–241, 1978.

[89] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[90] T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2 :80–94, 1987.

[91] Gadi Taubenfeld. Weak read/write registers. In Davide Frey, Michel Raynal, Saswati Sarkar, RudrapatnaK. Shyamasundar, and Prasun Sinha, editors, *Distributed Computing and Networking*, volume 7730 of *Lecture Notes in Computer Science*, pages 423–427. Springer Berlin Heidelberg, 2013.

[92] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[93] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

[94] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1) :40–44, January 2009.

[95] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt : Combining acid and base in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2014), OSDI*, volume 14, pages 495–509, 2014.

Glossaire

Ce glossaire contient un certain nombre de mots-clés et de notations utilisées dans ce document. Une brève description est fournie, ainsi qu'une liste des pages significatives où ce mot-clé/notation est utilisé.

$1\mathbf{W}n\mathbf{R}$ *Voir mono-écrivain multi-lecteurs.* 10, 14, 16, 27, 59, 91
V_E Ensemble des valeurs écrites pendant l'intervalle de temps $\mathcal{I}$. 38, 43, 52
V_L Ensemble des valeurs lues pendant l'intervalle de temps $\mathcal{I}$. 38, 43, 52
V_v Ensemble des vieilles valeurs lues pendant l'intervalle de temps $\mathcal{I}$. 38, 43, 52, 54
V_p Ensemble des valeurs présentes locament dans des processus au début de $\mathcal{I}$. ... 44, 52
V_m Ensemble des valeurs présentes uniquement dans des messages au début de $\mathcal{I}$. . 44, 52
U Ensemble des valeurs de V_m stockées localement sur un processus à un moment de $\mathcal{I}$. 45, 52
$\lfloor n/(n-f) \rfloor$ Nombre maximal de groupes de $n-f$ processus, deux à deux disjoints, que peut contenir un système. 59, 82, 83, 87, 89, 90, 95, 100
$\mathcal{P}$ L'ensemble des n processus du système. 8, 28, 67
$\perp$ Valeur par défaut d'un registre. 9, 38, 64
f Borne sur le nombre maximal de processus susceptibles de tomber en panne au cours d'une exécution. 11
id_i L'identifiant du processus p_i. Il s'agit d'un entier naturel associé de manière unique à p_i. 8, 62, 69, 77, 82, 84, 89, 91
n Nombre de processus du système. 8
$n-f$ Nombre maximal de processus garantis d'être corrects, et donc nombre maximal de réponses que peut attendre un processus. 18, 20
p_i Le i-ème processus du système, d'après un ordre arbitraire. $i \in [1, n]$. 8
seq Numéro de séquence permettant d'identifier si un message reçu est bien associé au message envoyé précédemment dont il est supposé être la réponse. 17, 40, 48
ts *Voir estampille.* 17

algorithme distribué Ensemble de n algorithmes locaux composés de calculs locaux et de communications. *Défini page 12.* 12
algorithmique distribuée Étude d'algorithmes et de problèmes dans un système distribué. 1
asynchrone Tel que les communications peuvent prendre un temps non borné. *Défini page 11.* 2, 11, 13

banc de registres χ-colorés Ensemble de χ registres, dont au moins un vérifie la propriété de disponibilité. *Défini page 27.* 27, 30, 33, 59
barrière d'une majorité de pannes Métaphore symbolisant la grande différence de possibilités entre le modèle dans lequel seule une stricte minorité des processus est susceptible de tomber en panne, et celui dans lequel il est possible que la majorité des processus tombent en panne. 4, 5, 16, 21, 36, 59
bruyant Se dit d'un algorithme dans lequel des messages sont constamment envoyés entre les processus, même lorsque l'état du système n'évolue pas. 39

byzantin Type de défaillance pour lequel un processus prend un comportement arbitraire. 11, 13

canal Lien entre 2 processus p_i et p_j permettant à p_i d'envoyer des messages à p_j. 9, 13, 44, 97

CAP Théorème énonçant qu'il est impossible de garantir à la fois les 3 propriétés Cohérence, Disponibilité, et Tolérance au partitionnement. *Voir [54] et page 3.* 3, 20

capturé Un diviseur réutilisable est capturé si un processus a obtenu *stop* mais pas encore appelé *Relache*. 74, 90, 94

capturé Une valeur x est capturée au cours d'un renommage si un processus l'a obtenue et pas encore libérée. 77, 89, 98

cohérence Propriété globale que doivent vérifier les données (valeurs) du système. 3, 14, 20, 21, 36, 59

cohérence causale forme de cohérence faible conservant l'ordre des liens de causalité entre opérations. 23

cohérence finale Propriété sur des objets de type registre, garantissant une stabilisation du système. Si à partir d'un certain temps aucune écriture n'a lieu, à partir d'un autre instant, toute lecture retourne la même valeur. 22, 37

concurrent Se dit d'intervalles de temps (ou d'opérations prenant un certain temps) ayant une intersection non-vide. *Défini page 12.* 12, 16, 38

correct Un processus est correct au cours d'une exécution s'il ne tombe pas en panne durant cette exécution. 11, 18, 27, 38, 62, 68, 74, 77, 83, 90

couleur Entier c représentant l'indice d'un registre χ-coloré dans son banc. . 28, 30, 32, 34

disponibilité Aussi appelée terminaison. Propriété affirmant que les requêtes et opérations se déroulent en un temps fini. 3, 22, 27, 36

diviseur Objet de mémoire partagé. Tout processus y accédant via une opération *Divise* obtient une direction dans $\{bas, droite, stop\}$. 6, 25, 71, 78

à-un-coup Ne peut être utilisé qu'au plus une fois par processus. *Défini page 68.* 68, 71

réutilisable Peut être utilisé plusieurs fois, et dispose d'une nouvelle opération *Relache*. *Défini page 73.* 73, 78

k-diviseur Version étendue du diviseur, pour lequel au plus k processus obtiennent *stop* (au lieu d'au plus 1). 83, 88, 90, 95

à-un-coup Ne peut être utilisé qu'au plus une fois par processus. *Défini page 83.* 83, 88

réutilisable Peut être utilisé plusieurs fois, et dispose d'une nouvelle opération *Relache*. A aussi une propriété affaiblie par rapport au diviseur réutilisable classique. *Défini page 90.* 90, 95

défaillance Défaut du système pouvant survenir de manière imprévisible. *Voir panne, byzantin.* 2, 11, 13

écriture Opération sur un objet partagé de type *registre* qui modifie l'état de ce registre en y écrivant une nouvelle valeur. 9, 16, 27, 38, 54

écrivain Processus pouvant utiliser des opérations d'écriture sur un registre 1WnR.. . . . . 16, 27, 38

envoi de messages Interface de communication dans laquelle les processus communiquent les uns avec les autres en s'envoyant directement des messages. *Défini page 9.* 2, 9, 12, 16

espace de noms Ensemble des valeurs de retour possibles pour un algorithme de renommage, généralement noté $[0, M-1]$ ou $[1, M]$. 25, 62, 68

estampille Entier représentant l'âge relatif d'une valeur. On écrira $\langle ts, v \rangle$ un couple estampille-valeur, et une estampille plus élevée représente une valeur plus récente.. . . . . 17, 37, 91

exclusion mutuelle Problème distribué dans lequel un et un seul processus doit obtenir une réponse positive. 25, 68

exécution Comportement possible d'un système distribué suivant un algorithme, mais pouvant être victime de défaillances et d'asynchronicité. *Défini page 12.* 12, 18, 27, 38, 54, 82, 83, 90

FIFO Caractéristique d'un canal de communication signifiant que les messages envoyés par ce canal sont reçus dans le même ordre qu'ils ont été envoyés. . . 9, 13, 45, 97

fonction de coloration de quorums Fonction qui à un quorum Q de $n - f$ processus associe un entier c, tel que 2 quorums de même couleur aient une intersection non vide. *Défini page 29.* 29, 30, 33, 59

grille de diviseurs Ensemble d'objets de type diviseur, ordonnés sous forme de grille à 2 dimensions. Chaque objet de la grille a des coordonnées $\langle a, b \rangle$ avec $a, b \in \mathbb{N}$. 71, 78, 88, 95, 98

indistinguable Impossible de faire la différence entre deux situations. Se réfère généralement à deux exécutions possibles, du point de vue d'un processus. . . . 13, 55, 58, 83, 84

lecture Opération sur un objet partagé de type *registre* supposée retourner la dernière valeur écrite dans ce registre. Ses propriétés varient selon le type de registre. 9, 14, 16, 27, 37

linéarisabilité Propriété représentant le fait que, pour toute exécution, chaque opération peut, d'un point de vue logique, être vue comme n'ayant pris qu'un instant. *Défini page 10.* 10, 16, 27

localement correct Se dit d'un algorithme tel que tout processus p_i vérifie que les valeurs successives de ses variables forment une suite croissante. 64, 87

majorité/minorité de pannes Se réfère aux cas $f \geq n/2$ et $f < n/2$, c'est-à-dire la possibilité (ou non) qu'une majorité des processus du système tombe en panne au cours d'une exécution. 3, 16, 20, 27, 61

modèle Définit les différents aspects d'un système distribué considéré, comme par exemple l'interface de communication ou le type de défaillances. 2, 8, 13, 17

modèle considéré / notre modèle Modèle asynchrone avec des pannes. Sauf mention contraire, les communications se font par envoi de messages dans un graphe complet de canaux sûrs. *Voir page 13.* 2, 13

mono-écrivain multi-lecteurs Se dit d'un registre pour lequel un seul processus peut utiliser l'opération d'écriture, alors que tous les processus peuvent utiliser celle de lecture. 10, 14, 16, 27, 38, 74

mémoire partagée Interface de communication constituée d'objets globaux auxquels les processus peuvent accéder via des opérations. *Défini page 9.* 2, 9, 12, 16, 25

obsolescence bornée Propriété de cohérence sur des objets de type registre, limitant les valeurs retournées par une lecture à l'une des k dernières valeurs écrites. . . 21, 37

occupé Un diviseur réutilisable est dit occupé si une opération est en cours, ou si un processus l'a capturé. *Défini page 74.* 74, 90, 96

panne Type de défaillance. Un processus qui tombe en panne cesse définitivement toute activité. *Défini page 11.* 2, 11, 13

en parallèle Un processus exécute plusieurs fonctions/algorithmes *en parallèle* s'il effectue simultannément ou alternativement des pas de calcul dans chacun de ces fonctions/algorithmes. En particulier, dans le cas ou un de ces fonctions/algorithmes est bloqué dans une attente infinie, les autres continuent de progresser. . 12, 28, 34

partitionnement Situation dans laquelle le système distribué est divisé en plusieurs partitions (ensembles de processus) distinctes, chacune coupée des autres. . . 3, 22

problème Ensemble de contraintes que l'on souhaite résoudre par un algorithme dans un système distribué. *Défini page 12.* 12, 24, 34

processus Entité capable d'effectuer des calculs, de stocker des données, et de communiquer avec d'autres processus via une interface de communication. *Défini page 8.* 1, 8

précède Se dit d'un intervalle de temps dont la fin se situe avant le début d'un autre intervalle de temps (et de même pour deux opérations). *Défini page 12.* . 12, 16, 38, 70

quorum Ensemble des processus ayant participé à une opération, généralement en répondant à un message de requête. Comprend $n-f$ processus. *Voir page 17 pour un exemple de contexte.* 17, 20, 29, 40

registre Objet de mémoire partagé, stockant une valeur v. Les processus peuvent modifier cette valeur via une opération Ecriture et la récupérer via une opération Lecture. *Défini page 9.* 3, 6, 14, 16, 27, 36, 69, 74

- **atomique** Registre vérifiant le critère de cohérence le plus fort : la *linéarisabilité*. *Défini page 10.* 10, 16
- **régulier** Registre tel que toute opération de lecture retourne la dernière valeur écrite ou une valeur écrite concurremment. *Défini page 14.* 14
- **sûr** Registre tel que toute opération de lecture concurrente avec une opération d'écriture peut retourner n'importe quelle valeur. *Défini page 14.* 14

registre χ-coloré Objet similaire au registre atomique, mais ne vérifiant pas (seul) la propriété de disponibilité. *Voir banc.* 27, 30

α-registre Registre pouvant stocker jusqu'à α valeurs à la fois, mais ces valeurs ne sont pas choisies et pas nécessairement les dernières écrites. *Défini page 38.* . 38, 39, 58

renommage Problème dans lequel les processus cherchent à acquérir une valeur unique.. . . . . 6, 25, 61, 71, 82

- **à-un-coup** Chaque processus ne peut obtenir qu'une valeur. *Défini page 61.* . . 62, 71
- **réutilisable** Chaque processus peut obtenir une valeur, la libérer, et recommencer plusieurs fois. *Défini page 77.* 77

M-renommage Renommage dans un espace de nom de taille M. 62, 73, 81

- **non-trivial** Tel que la taille M de son espace de noms ne dépend pas de la taille des identifiants des processus. 82, 89

renommage k-redondant Version étendue du renommage, pour lequel au plus k processus obtiennent une même valeur (au lieu d'au plus 1). 82, 89

- **à-un-coup** Chaque processus ne peut obtenir qu'une valeur. *Défini page 82.* 82
- **réutilisable** Chaque processus peut obtenir une valeur, la libérer, et recommencer plusieurs fois. *Défini page 89.* 89

(M,k)-renommage Renommage k-redondant dans un espace de nom de taille M. . 82, 89

silencieux Se dit d'un algorithme dans lequel un état stable du système (l'état des processus n'évoluent pas) finit par signifier une absence de messages en cours. . 39, 49

simulation ABD Simulation de registres atomiques dans un modèle par envoi de messages, asynchrone et avec une minorité de pannes, issue de [16]. *Détaillée dans la partie 3.1.* 16, 30, 37

snapshot Opération sur un vecteur de registres, permettant de lire l'ensemble des valeurs de façon instantanée d'un point de vue logique. 5, 28, 59

stable Se dit d'un tableau V qui a été reçu par un processus p_i de $n - f - 1$ autres processus, tout en gardant ce tableau dans sa variable locale $V_i = V$. 64, 87

synchrone Tel qu'une borne sur les temps de communication est connue. 11

système distribué Système composé de processus indépendants et dotés d'une interface de communication. 1, 8, 54, 82, 89

sûr (canal) Caractéristique d'un canal de communication signifiant que les messages envoyés par ce canal seront reçus intacts par leur destinataire. 9, 13, 20